AFRL-IF-WP-TR-2004-1516

INTEGRATED MANAGEMENT OF POWER AWARE COMPUTATION AND COMMUNICATION (IMPACCT)

Dr. Pai Chou and Dr. Nader Bagherzadeh

University of California, Irvine
160 Administration
Irvine, CA 92697-1875

MAY 2003

Final Report for 08 May 2000 – 07 May 2003

Approved for public release; distribution is unlimited.

STINFO FINAL REPORT

INFORMATION DIRECTORATE
AIR FORCE RESEARCH LABORATORY
AIR FORCE MATERIEL COMMAND
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7334

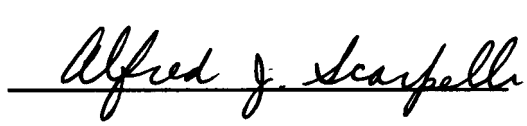
NOTICE

USING GOVERNMENT DRAWINGS, SPECIFICATIONS, OR OTHER DATA INCLUDED IN THIS DOCUMENT FOR ANY PURPOSE OTHER THAN GOVERNMENT PROCUREMENT DOES NOT IN ANY WAY OBLIGATE THE US GOVERNMENT. THE FACT THAT THE GOVERNMENT FORMULATED OR SUPPLIED THE DRAWINGS, SPECIFICATIONS, OR OTHER DATA DOES NOT LICENSE THE HOLDER OR ANY OTHER PERSON OR CORPORATION; OR CONVEY ANY RIGHTS OR PERMISSION TO MANUFACTURE, USE, OR SELL ANY PATENTED INVENTION THAT MAY RELATE TO THEM.

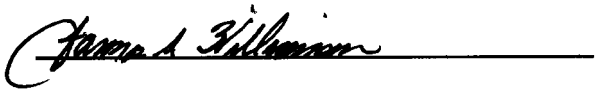
THIS REPORT IS RELEASABLE TO THE NATIONAL TECHNICAL INFORMATION SERVICE (NTIS). AT NTIS, IT WILL BE AVAILABLE TO THE GENERAL PUBLIC, INCLUDING FOREIGN NATIONS.

THIS TECHNICAL REPORT HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION.

KERRY L. HILL
Project Engineer
Embedded Info Sys Engineering Branch
Information Systems Technology Division



ALFRED J. SCARPELLI
Team Leader
Embedded Info Systems Engineering Branch
Information Systems Technology Division



JAMES S. WILLIAMSON, Chief
Embedded Info Systems Engineering Branch
Information Systems Technology Division
Information Directorate

Do not return copies of this report unless contractual obligations or notice on a specific document requires its return.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
May 2003	Final	05/08/2000 – 05/07/2003

4. TITLE AND SUBTITLE
INTEGRATED MANAGEMENT OF POWER AWARE COMPUTATION AND COMMUNICATION (IMPACCT)

5a. CONTRACT NUMBER
F33615-00-1-1719

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER
69199F

6. AUTHOR(S)
Dr. Pai Chou and Dr. Nader Bagherzadeh

5d. PROJECT NUMBER
ARPI

5e. TASK NUMBER
FT

5f. WORK UNIT NUMBER
0K

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

University of California, Irvine
160 Administration
Irvine, CA 92697-1875

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Information Directorate
Air Force Research Laboratory
Air Force Materiel Command
Wright-Patterson AFB, OH 45433-7334

DARPA/IPTO
3701 Fairfax Drive
Arlington, VA 22203-1714

10. SPONSORING/MONITORING AGENCY ACRONYM(S)
AFRL/IFTA

11. SPONSORING/MONITORING AGENCY REPORT NUMBER(S)
AFRL-IF-WP-TR-2004-1516

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
The IMPACCT, or Integrated Management of Power Aware Computation and Communication, program objectives are to enhance the power/performance tradeoff range and to correctly compose different component level power management techniques at the system level. Power and timing constraints can be used as knobs to tune the system for performance or power, without hardwiring to either goal. To maximize performance and resolve power "hot spots," we exploit system-level task motion under pair-wise timing and total power as constraints. A distinguishing feature of our work is our ability to handle co-activation, an essential property for the correct operation of these embedded systems. Furthermore, we propose mode selection as a generalized way for fully exploiting novel power management features provided by an increasingly intelligent class of power-aware components. They are capable of managing power and provide many more power modes. However, today's power management techniques often cannot take full advantage of these rich features, but instead they use only two or three modes (e.g., on/off). Our mode selection methodology models the dependency and produces a mode schedule that considers restricted transitions and overhead amortization.

15. SUBJECT TERMS
Power Aware Computing and Communications (PAC/C), power management, power scheduling, voltage scaling

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT:	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON (Monitor)
a. REPORT	b. ABSTRACT	c. THIS PAGE			Kerry L. Hill
Unclassified	Unclassified	Unclassified	SAR	246	19b. TELEPHONE NUMBER *(Include Area Code)* (937) 255-7698 x3604

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39-18

Acknowledgement

This research was sponsored by DARPA under contract F33615-00-1-1719. It represents a collaboration between the University of California at Irvine and the NASA/Cal Tech Jet Propulsion Laboratory. Special thanks to Dr. N. Aranki, Dr. B. Toomarian, Dr. M. Mojarradi and Dr. J. U. Patel at JPL and Kerry Hill at AFRL for their discussion and assistance.

Contents

List of Figures

List of Tables

Part I

Introduction and Overview

Chapter 1

Introduction

Recent years have seen the emergence of *power-aware* embedded systems. They are characterized by not only low power consumption, but more generally by their ability to support a wide range of power/performance trade-offs. These systems can be viewed as providing "knobs" that can be turned one direction to reduce power consumption or the other direction to increase performance. The ability to maximize the range of power-performance trade-offs is driven by new applications that demand very high performance while operating under stringent timing and power constraints. One such application can be found in the space domain in the form of a rover.

Let us consider the Mars Pathfinder rover from NASA/JPL [4]. It was designed to roam on Mars to take digital photographs and perform scientific experiments over several hundred days. Its energy sources consist of a battery pack and a solar panel, and future versions are expected to incorporate a nuclear generator or other energy scavenging devices. The initial version was designed to be low-power, and this was accomplished by serializing all tasks, including mechanical and heating as well as computation. However, low-power also means low performance in this case, as the rover could move at most 10cm per minute, and shoot and wirelessly transmit at most three

high-resolution photos in a day. Even though during daytime the solar panel could output more power than could be consumed by the rover, the rover was unable to take advantage of this power; instead, the extra heat was redirected to heating the wheels.

This is an instance where a low-power design may be correct, but a power-aware version can do much better. We have proposed a power-aware version of the rover: by allowing power usage and performance to track power availability, the power-aware system with more system-level parallelism achieved 33% speedup while saving 33% battery energy [44].

Encouraged by the initial success, we explored additional power management opportunities at the system level. Since the goal is to increase the dynamic range of power/performance curves, we sought ways to increase performance in one direction and to reduce power in the other. To increase performance when more power (such as solar) is available, we attempted system-level task motion, a class of effective techniques that have been developed for many different domains ranging from VLIW instruction scheduling to hardware synthesis. To reduce energy consumption, we also attempted to incorporate other researchers' new power management techniques that are power-aware. These include a variety of dynamic voltage scaling (DVS) and scheduling algorithms for modern embedded processors, whose voltage and frequency can be controlled.

However, a somewhat surprising result was that many of these performance enhancement and power reduction techniques yield incorrect and rather counterintuitive results when applied together at the system level. Existing scheduling techniques that treat the power budget as a resource constraint (e.g., mapping power to the total register count) fail to correctly satisfy the power constraints. On the other hand, DVS techniques, which slow down processors in order to achieve quadratic energy savings, actually end up consuming more energy at the system level.

The main reason these techniques fail is that many important system-level depen-

dencies are not properly modeled or considered. In a system, the components do not work independently; instead, they work very much together with each other, and power management decisions made on one component can have a chain of effects on the power usage of the other components. This is further complicated by the fact that different components are built with different power management capabilities. In the Mars rover, not all components are power manageable. In fact, some components include motors for steering and driving the rover, heating elements for melting the frozen lubricants on the wheels, and the R/F module. Many of these components cannot scale their voltage or frequency the same way a processor can. Furthermore, mode changes are seldom instantaneous or free; instead, they incur nontrivial timing and power overhead that cannot always be amortized. As a result, the combined effect of these power management techniques can often contradict the designer's intuition and even cancel each other's effects.

It is clear that an integrated design tool is sorely needed to help designers manage such a multi-dimensional problem: functional correctness, timing constraints, and power awareness. To address these difficult problems, we develop a tool-based design methodology called IMPACCT, for Integrated Management of Power-Aware Computing and Communication Technologies. As with most system-level design tools, IMPACCT starts with high-level modeling of the application, separate from the target architecture. The designer then uses IMPACCT to transform and refine the high-level model towards implementation. IMPACCT also supports power-aware functional simulation to help with design validation.

This report focuses on two of the core design tasks in IMPACCT: power-aware scheduling and mode selection. The objectives are to enhancing the power/performance trade-off range and to correctly compose different component level power management techniques at the system level. Power and timing constraints can be used as knobs to tune the system for performance or power, without hardwiring to either goal. To max-

imize performance and resolve power "hot spots," we exploit system-level task motion under pair-wise timing and total power as constraints. A distinguishing feature of our work is our ability to handle *co-activation*, an essential property for the correct operation of these embedded systems. Furthermore, we propose mode selection as a generalized way for fully exploiting novel power management features provided by an increasingly intelligent class of power-aware components. They are capable of managing power and provide many more *power modes*. However, today's power management techniques often cannot take full advantage of these rich features, but instead they use only two or three modes (e.g., on/off). Our mode selection methodology models the dependency and produces a mode schedule that considers restricted transitions and overhead amortization. Together these techniques not only form the foundation for integrating many power management techniques, but more importantly they help even experienced designers avoid many pitfalls with composing these components at the system-level.

This report is a compilation of research work done by the IMPACCT project. Its main contribution is that, by taking an integrated, global perspective in managing power, it addresses the pitfalls of many of today's local, component-level techniques. The rest of Part I presents an overview of IMPACCT capabilities, including specification, architecture, simulation, and power management techniques. Parts II and III present two important classes of scheduling techniques. Part II present general time-constrained, power-constrained scheduling, compile-time scheduling techniques, while Part III exploits specialization to data-regular applications. Part IV investigates more architecture details with techniques we call mode selection and topology selection.

Chapter 2

Overview of Capabilities

2.1 Impacct overview

IMPACCT is a system-level design tool for exploring power/performance trade-offs in hard real-time systems by means of power-aware scheduling and architectural configuration. The current implementation includes an interactive graphical tool for scheduling, mode selection, and an interface to a simulation back-end for integrated evaluation of the system under design. Amdahl's law applies to power as well as performance. That is, the power saving of a given component must be scaled by its percentage contribution to an entire system. Furthermore, a system in the broad sense includes not only computational components but also those in the non-computational domains (e.g., mechanical and thermal subsystems), which are equally important in many mission-critical applications. IMPACCT is the first tool to correctly address all of these system-level power management issues. Fig. 2.1 shows the main components of the IMPACCT framework, and this chapter will highlight each box in order. The combination of these features in IMPACCT presents a compelling design-time tool for engineers to explore a wide range of system-level power/performance trade-offs with confidence.

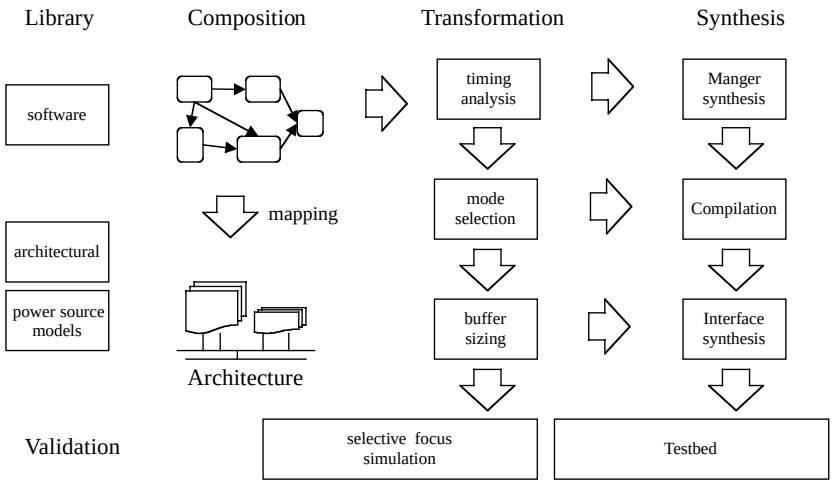


Figure 2.1: The IMPACCT system-level design tool for power-aware embedded systems.

2.2 Input: Application Model and Constraints

To use IMPACCT, the designer must construct a model for the application and constraints. Although the detailed application behavior is ultimately written in one of the system programming languages (such as C, C++, Ada, Java, etc), IMPACCT does not process these files directly; instead, they are passed to power/timing analysis or simulation tools for estimation or validation. IMPACCT expects the designer to construct a higher-level model for the application in our custom language. This description includes ports and channels for expressing data dependencies, and it supports timing and power constraints. Note that timing and power are not necessarily intrinsic to the application problem itself, but they should really be viewed as "budgets" whose values are selected based on engineering decisions. One main purpose of the tool is to help designers with constraint refinement or adjustment (re-budgeting) by giving them a quick estimate. This approach allows the designer to start working with the power/timing budget for various tasks to be performed long before the program or component is designed. As these pieces become available, they will then be used to refine these budgets

with more accurate estimation.

We currently support power constraints and timing constraints. *Power constraints* are the min/max bounds on the power dimension of the power profile. The *max-power* constraint requires that the system never draw more than the specified amount of power at any given moment. It may be derived from the maximum current rating of the power supply and can be a hard constraint. Even though most systems to date could assume sufficient power by design, the next generation power-aware embedded systems will need to work with a much more diverse set of power sources with much lower power budgets and reduced availability. This will make max-power a hard constraint. On the other hand, we believe *min-power* will be an important constraint, especially for systems with renewable energy sources. One reason is that as heat becomes an important issue in embedded systems, unused power must be carefully dissipated or else the system risks overheating. Rechargeable batteries have finite capacities and will contribute to the heat when overcharged. In the case of the rover, it would require extra heat dissipation hardware to handle unused min-power. This would add extra weight and cost to the rover. Another reason is that the min and max constraints together will be a way to explore power/performance trade-offs without being hardwired to the low-power goal. Both min and max power constraints may be functions over time.

Timing constraints are in the form of *min/max timing separation* between pairs of events, where an event can be the start or end of a task. This is a general way for expressing precedence, absolute and relative deadlines, and also co-activation. Tasks assigned to different resources may run in parallel. We currently use a simple custom language to capture these timing constraints. The syntax of this high-level file is not important; it just has to be expressive enough to construct a graph description of the pair-wise timing constraints.

2.3 Target Architecture and Mapping

The input to IMPACCT consists of a model for the target architecture and application-to-architecture mapping. The target system architecture provides the primitives for power management as well as the power/timing attributes needed for scheduling and mode selection. The elements of the application model are mapped to those of the target architecture: that is, the tasks are mapped to the processors, and channels mapped to the busses. IMPACCT provides a *component library* and a *system architecture template* to aid the description of the target architecture.

The component library consists of models for components and busses in the target architecture. They include processors, memory modules, bus controllers, communication modules, sensors and actuators, digital cameras, and various peripheral devices. The designer instantiates and configures these components from the library. The component models will provide an interface for the rest of the design tool to ask questions about the power/timing attributes needed to synthesize or for simulation. Some of these attributes such as modes, clock rates, or voltage may be stored as fixed values, but others such as the execution delay or the power consumption may need to be derived by either evaluating a formula or by simulation. Each component model may encapsulate any number of detailed models (RTL, SPICE, power-macromodel), but they are abstracted from the designer. IMPACCT augments these low-level models with higher-level models for supporting system-level power management. These features include the power modes, the allowed transitions between modes, the power/timing coefficients associated with each mode and the transitions, and the interface description for controlling these power management features. This mode model will be described in more detail in the Mode Selection chapter.

Unlike traditional hardware/software co-design that is more about free-form exploration of an optimal architecture, we take a platform-based approach for practical reasons. IMPACCT provides architectural templates for configurable platforms, and

currently supported is a symmetric multiprocessor architecture interconnected with a two-tier bus. It can be configured for different numbers of processors and components from the library. The two-tier bus includes the IEEE 1394 (“FireWire”) for high-speed, real-time data and the I^2C for low-speed control. Both are power efficient and support dynamically adding/removing or powering up/down individual nodes for the purpose of power management. The software runs on WindRiver vxWorks, a commercial real-time OS for embedded systems. Scheduling and mode selection are performed statically but the run-time system can switch between different schedules.

2.4 Power-Aware Scheduling

Our scheduler enhances the dynamic range of power/performance trade-offs. The core scheduler handles both timing and power as constraints, not just goals. Power and timing are both treated as min/max constraints. The advantage is that these constraints become the knobs for tuning the system’s power/performance trade-offs. By making the constraints track the available solar power, the IMPACCT scheduler has been shown to accelerate the system while saving energy at the same time for a Mars rover. This feature will be critical to also systems that use alternative energy sources such as thermal batteries as well as those with thermal management concerns. In addition, we explore system-level task motion as a way to vary the level of parallelism to further increase the dynamic range of these systems. Scheduling will be discussed in Chapter 3.

2.5 Mode Selection

Another complementary feature in IMPACCT is mode selection. It is the task of deriving a schedule for mode changes in the components of the system, such that all architectural effects are properly considered. It takes as input a schedule from the previous step, and it decides what power modes in which each component should operate

over time. Mode selection addresses issues that fail to be handled by today's greedy dynamic voltage schedulers by considering the transition overhead and dependencies. It will not change mode if the time/power overhead involved cannot be amortized over the tasks to be performed, and it also prevents system-level power spikes due to greedy, isolated voltage scaling. More importantly, IMPACCT's mode selection properly models and handles co-activation dependencies. For example, when the processor is on, the memory must be on, too. By modeling these dependencies in the mode selection step, IMPACCT will ensure that the resulting power management policy considers all features critical to the correct operation of the entire system. Mode Selection will be described in more detail in Chapter 7.

2.6 Simulation Support

IMPACCT supports simulation at various stages of the design flow. The high-level application description can be simulated functionally without mapping to an architecture. The IMPACCT high-level simulator has been integrated into the scheduler. It not only computes the ordering of the tasks to run on generic resources, but also invokes the compiled application files via native calls to simulate their functionality. The high-level simulator is also responsible for implementing the inter-process communication mechanisms using buffer management.

This setup also enables the integration of heterogeneous simulation and emulation models with a uniform user interface. Because the simulation models are externalized, the IMPACCT simulation coordinator can replace the external, native calls with any other calls, as long as they conform to a compatible application programming interface. For example, hardware-in-the-loop simulation can be accomplished by replacing these external calls with calls to device drivers that control emulation hardware. Similarly, these calls can also be made to detailed simulation models when accuracy or controllability is required. The back-end is completely decoupled from the front-end,

which provides a uniform user interface including visualization support.

Part II

General Scheduling

Chapter 3

Power Aware Scheduling

Power-aware systems are those that must make the best use of available power. They subsume traditional low-power systems in that they must not only minimize power when the budget is low, but also deliver higher performance when required. This chapter presents a new scheduling technique for supporting the design and evaluation to a class of power-aware systems in mission critical applications. It computes a schedule that satisfies stringent min/max timing and max power constraints at all times. Furthermore, it also makes the best effort to satisfy min power constraint in an attempt to fully utilize free solar power or to control power jitter. Experimental results show that our automated technique yields designs that improve performance and reduce energy cost simultaneously compared to hand-crafted designs used in previous missions. This tool forms the basis of the IMPACCT system-level framework that will enable designers to aggressively explore many more power-performance trade-offs with confidence.

3.1 Introduction

Power management is becoming one of the central issues in embedded systems. They are particularly critical to systems that must carry their own power source and cannot

rely on a power outlet on the wall. Without power, the system is useless. In the consumer space, the consequence may mean not being able to make an emergency call or other minor inconveniences; but in mission-critical systems, such a failure can cost millions and even human lives.

This chapter investigates key issues in power management for mission-oriented systems. Our motivating example comes from the NASA Mars Pathfinder rover developed at JPL [4]. It features several interesting properties that were not adequately addressed by previous work. First, such a system must be designed to be power-aware, rather than low-power. Second, it is critical that power management decisions be made at the system level, rather than only at the component level.

3.1.1 Power-aware vs. low-power

Traditionally, many components and systems have been designed to be low-power. However, we believe there is a critical difference between power-aware and low-power systems. Power-aware systems must make the best use of their available power, and they subsume low-power as a special case.

In the Mars rover case, its designers constructed a low-power design. It incorporated some of the best low-power design techniques at all levels of abstraction. The rover has two power sources: a solar panel and a non-rechargeable battery. To strictly control power draw, the designers serialized all tasks, including driving, steering, obstacle detection, and heating motors. This low-power design allows the rover to operate for hundreds of days during daylight, and it sleeps at night. However, full serialization also means the rover moves as slowly as 10cm per minute, and it can only take a total of three pictures per day.

A power-aware design can greatly improve the utility of the rover. We observe that the battery is non-rechargeable, and thus solar power would be wasted if not used while it is available. In the existing design, the rover follows the same serial schedule

regardless of the solar power level, and simply directs the excess energy to heating the wheels. A rover with more parallelism in its schedule can perform better (more tasks, more quickly) while saving even more battery energy than the existing low-power design if it can take advantage of the free power, as validated by our experiments in the results section.

3.1.2 System-level power-aware design

We believe that power-aware designs must be done at the system-level, not just at the component level. Amdahl's law applies to power as well, not just performance. That is, the power saving of a given component must be scaled by its percentage contribution in an entire system. If a component only draws 2% of the power in a system, a 50% reduction in its power amounts to merely 1% saving to the system. Therefore, it is critical to identify where power is being consumed in the context of a system, not just the components in isolation.

In the case of the Mars rover, it turns out that some of the biggest consumers are not even in the digital computer, but they also include the wheel motors, the steering motors, laser-guided obstacle detection, and the heaters. A successful power-aware design must consider these non-computation domains and coordinate their power usage as a whole system.

3.1.3 Approach: design tools

Our approach is to support power-aware design with a system-level design tool. One of the lessons learned from the Mars rover was that, without a tool, the designer had no choice but to embed many power-management decisions in the implementation. As a result, they were forced to design conservatively and could not consider more than one or two design alternatives. The purpose of our tool is to enable the exploration of many more points in the design space, so that additional knowledge about the mission

can be incorporated to refine the design without requiring dramatic redesign.

The work presented in this chapter represents one of the core tools in a larger design framework, called IMPACCT. The designers input a high-level behavioral specification of the design in terms of communicating processes and constraints. These processes have been assigned to run on specific execution resources, either interactively or semi-automatically by the design tool. The scheduling tool in this chapter constructs a constraint graph and performs power-aware scheduling. The output is then fed to another tool that performs optimizations and synthesis of power managers at the architectural level.

This chapter is organized as follows. Section 3.2 reviews related work, and Section 3.3 describes the application example in more detail. We present the problem formulation in Section 3.4 and graph-based scheduling algorithms in Section 3.5. Then, we discuss experimental results in Section 3.6 followed by our concluding remarks and future work.

3.2 Related Work

Prior works have addressed minimization of power usage at the system level. Their common goal is to minimize power usage while maintaining a satisfactory level of performance or meeting real-time constraints. However, these low-power techniques often cannot be directly adapted in power-aware systems.

3.2.1 Subsystem shutdown

Shutting down idle subsystems such as network interfaces, hard disks, and displays can save a significant amount of power in a system. The shutdown decision can be based on idle times of individual subsystems, although such approaches are less than satisfactory. Proposed improvements either attempt to make the timeout adaptive to the

actual usage pattern, or use profiling to help predict the proper time to shutdown and power up subsystems [62, 20, 64].

While it is important to manage the power of subsystems, unfortunately these techniques have several limitations. First, they do not handle timing constraints, including deadlines and min/max separation. Second, they are not power-aware in the sense that they do not distinguish between free power (such as solar sources) vs. expensive power (non-rechargeable battery). These power managers do not control their workload; instead, they make the best effort to minimize power consumption by treating the workload as a given.

3.2.2 Real-time scheduling

Many real-time scheduling techniques have been proposed to date, but only recently have researchers started to address power issues with the objective of minimizing power usage. For example, rate-monotonic scheduling has been extended to scheduling variable-voltage processors. The idea is to save power by slowing down the processor just enough to meet the deadlines [48].

Such techniques have several limitations. First, they are CPU schedulers that minimize CPU power, rather than power managers that control subsystems and task executions. Second, in practice, it is difficult to tune the voltage or frequency scale to such a fine precision. As a result, the risk of missing deadlines may be high, even if context switching overhead is taken into account. Also, while these schedulers meet timing constraints, they do not handle constraints on power usage.

3.2.3 Power awareness

We believe power-aware scheduling must have several key features. First, they must handle both timing and power stringently as hard constraints. This is unlike previous work that treats them as desirable by-products but cannot always make strong guar-

antees. Second, domain-specific knowledge about the power source, battery model, and other operating conditions must be expressible in terms of supported types of constraints on the timing and power. The types of constraints that are sufficiently expressive for our application are min and max timing constraints on tasks, as well as min and max power constraints on the system. Min/max timing constraints subsume deadlines and precedence dependencies and can express dependencies across subsystems [18, 19]. Max power would track the budget imposed by the current power sources. Min power constraints, strictly speaking, may be counter-intuitive in that it forces the power manager to maintain a certain level of activity. The primary motivation is that power from solar panels or other free sources that cannot be stored should be fully utilized greedily, or else they will be wasted. Another motivation is to control the jitter in the system-level power curve in an attempt to optimize battery usage. However, min power constraints are not imperatively enforced, and we assume that they may be violated occasionally or be met by scheduling background tasks.

3.3 Motivating Example

To demonstrate the effectiveness and applicability of our power-aware scheduling techniques, we choose the NASA/JPL Mars rover as our motivating example. Its mission is to perform scientific experiments and imaging on Mars surface. The rover is deployed and operated for at least 7 sols (days on Mars). If it keeps performing well at the end of the designated period, an extended mission may continue. The rover's power sources consist of a non-rechargeable battery pack and a solar panel. Clearly, the duration of a mission is limited by the amount of remaining battery energy. Thus, a careful management of power usage may yield potential energy savings, as well as performance speedup.

The rover travels between different target locations before experiments and imaging can be performed. Since the temperature on Mars surface can be as low as -80°C,

Operation	Duration(s)	Timing constraints
Heating steering motors	5	At least 5s, at most 50s before steering
Heating wheel motors	5	At least 5s, at most 50s before driving
Hazard detection	10	At least 10s before steering
Steering	5	At least 5s before driving
Driving	10	At least 10s before next hazard detection

Table 3.1: Timing constraints in Mars rover's operations.

driving in low temperature requires more power and energy consumption because the motors must be heated periodically. This fact indicates that mechanical and thermal subsystems are the major power consumers. Therefore, our model targets the mechanical and thermal subsystem under a typical mission scenario when the rover is moving to the next location.

We give a high-level description of the rover's operations. The rover drives about 7cm in distance in one single step of movement. During each step, it must first detect any obstacles in the moving direction and choose a safe angle to move. Then the four steering motors are started to turn to the right direction. Finally, the six wheel motors are driven to perform one step of movement. Therefore, hazard detection - steering - driving must operate in sequence. The other set of timing constraints comes from the requirement to heat the motors before steering and driving. All four steering motors and six wheel motors must be heated within a certain period prior to mechanical operations. The timing constraints are summarized in Table 3.1. The power consumption of each operation varies with environmental temperature. We assume that the temperature is closely related to the sunlight density that can be measured by power output from the solar panel. In order to examine how the power-aware scheduling techniques handle different constraints, we investigate three cases of solar power output: best case is 14.9W at noon time; the typical case is 12W; and the worst case is at dusk. The maximum supply power is limited by the threshold of battery power output, which we assume to be 10W. Therefore, in all cases, the rover can be safely operated only

Power sources & tasks	Duration (s)	Power (W)		
		Best case @-40°C	Typical case @-60°C	Worst case @-80°C
Solar panel		14.9	12	9
Battery pack		10 max	10 max	10 max
CPU	constant	2.5	3.1	3.7
Heating two motors	5	7.6	9.5	11.3
Driving	10	7.5	10.9	13.8
Steering	5	4.3	6.2	8.1
Hazard detection	10	5.1	6.1	7.3

Table 3.2: Power consumption of Mars rover's operations.

if its instantaneous power consumption is less than available solar power plus 10W maximum battery power output, which constitutes the max power constraint. We also extract the solar power level as the min power constraint to distinguish such free power from the costly power. Table 3.2 illustrates the power sources and consumers in three cases.

The goal of a scheduler is to assign tasks to time slots such that all timing and power constraints are satisfied. Without an automated tool, the existing solution by JPL had to be hand-crafted. It serializes all operations to minimize power draw from the non-rechargeable battery. The existing design is very low-power, but is also very slow and can possibly incur additional energy cost in some bad cases.

By introducing power-aware scheduling, not only could we improve performance, but also save non-rechargeable energy by better utilization of solar energy. This is in contrast to the conventional trade-off between energy and performance, where improvement on one is done at the expense of each other. A power-aware approach can win both at the same time. Section 3.6 provides a detailed analysis to a case study on the Mars rover example.

3.4 Problem Formulation

Our problem formulation is based on an extension to a constraint graph used in a previous time-driven scheduling problem [18]. Section 3.4.1 reviews the base formulation and our extension on parallel execution and slack properties. Section 3.4.2 defines power characteristics of the scheduling problem including the power profile of a schedule and new properties by applying the max and min power constraints. Section 3.4.3 presents a new way of viewing the time/power scheduling problem as a two-dimensional constraint problem by drawing analogies from the Gantt chart.

3.4.1 Constraint graph and properties

We formally define the concepts in our model as we construct the constraint graph formulation for a scheduling problem. These concepts include tasks, timing constrains, schedules and the slack properties of a schedule.

Definition 1 (Tasks $u \in T$) Given T as the set of all tasks and a set of execution resource R, a task $u \in T$ is characterized by a set of functions, $u = \{r(u), d(u), p(u)\}$, where $r(u) \in R$ is the execution resource onto which the task is mapped, $d(u)$ is its execution delay, and $p(u)$ is its power consumption.

To handle parallel execution resources that consume power, the function $r : T \rightarrow R$ maps each task to a resource set R. Examples of execution resources include not only computing resources such as an embedded microprocessor, but also other consumers of power, e.g. mechanical subsystems and heaters. We further assume that if two tasks u and v are mapped to the same resource ($r(u) = r(v)$), then u and v must be serialized in the final schedule to eliminate resource conflicts.

The execution of task u takes $d(u)$ time units. We also assume the availability of the power consumption function, $p : T \rightarrow \mathbb{R} > 0$, which returns the estimated power consumption by all tasks. As a result, the energy consumption of task u is $d(u) \times p(u)$.

In practice, the power consumption can be either in the form of (min, typical, max), or a function over time, rather than an exact value. Since our formulation can be extended to handling these cases, we will assume a single value to simplify the discussion.

Definition 2 (Timing constraints) A timing constraint specifies the timing relationship between two tasks $u, v \in T$, in one of the two forms:

(1) A *min timing constraint* $u \rightarrow v : \delta, \delta \geq 0$ indicates that v must start at least δ time units after u starts, formally $t_v - t_u \geq \delta$.

(2) A *max timing constraint* $u \leftarrow v : \delta, \delta > 0$ indicates that v must start at most δ time units after u starts, formally $t_v - t_u \leq \delta$.

A min timing constraint $u \rightarrow v : \delta$ implies task u precedes v, since $t_v - t_u \geq \delta \geq 0$; while a max constraint $u \leftarrow v : \delta$ does not imply any precedence relationship between u and v. This min-max timing separation handles more general timing relationships between tasks. For example, a task u with a deadline τ to finish its execution ($\tau \geq d(u)$) is a special case of a max timing constraint: $a \leftarrow u : \tau - d(u)$, where a, the *anchor*, is a virtual task that starts the schedule, $t_a = 0$.

Definition 3 (Schedule σ, Finish time τ_σ) Given a task set T,

(1) A *schedule* σ assigns start time t_u to every task $u \in T$. Without ambiguity, we further overload the σ notation to map any task u to its assigned start time according to σ, that is, $\sigma(u) = t_u$.

(2) The *finish time* of a schedule σ is the time when all tasks in T finish their execution. It is defined as $\tau_\sigma = \max(\sigma(u) + d(u)), \forall u \in T$.

We construct a constraint graph based on the tasks, their resource mapping and the corresponding constraints among the tasks in a scheduling problem. A schedule

as the solution to the problem can be computed based on the constraint graph and its properties.

Definition 4 (Constraint graph $G(V,E)$) Given a task set T, a resource set R to which all tasks in T are mapped, and a timing constraint set C specifying the timing constraints between tasks in T, a *constraint graph* $G(V,E)$ can be constructed as follows.

(1) The vertices V represent all tasks, $V = \{a\} \cup \{u | \forall u \in T\}$. Each vertex $u \in V$ has three attributes: $r(u), d(u), p(u)$ representing its resource mapping, execution delay and power consumption of task u, respectively. $r(a) = nil, d(a) = 0, p(a) = 0$.

(2) The edges $E \subseteq V \times V$ represent timing constraints between tasks. For two vertices $u, v \in V$, an edge (u,v) with weight $w(u,v)$ is denoted as $(u,v) : w(u,v)$. It specifies a timing constraint between task u and v, such that $t_v - t_u \geq w(u,v)$.

(2.a) A min timing constraint $u \rightarrow v : \delta, \delta \geq 0$ is represented by an edge $(u,v) : \delta$ with non-negative weight $\delta \geq 0$, called a *forward edge*.

(2.b) A max timing constraint $u \leftarrow v : \delta, \delta > 0$ is represented by an edge $(v,u) : -\delta$ with negative weight $-\delta < 0$, called a *backward edge*.

An example of a constraint graph is shown in Fig. 3.1. Nine tasks named $a \ldots i$ are mapped into three resources, A, B and C. Each vertex u is denoted with a name and its attributes in the form of $r(u)/d(u)/p(u)$.

Lemma 1 (Schedulability property) Given a scheduling problem with a task set T, a resource set R and a timing constraint set C formulated as a constraint graph $G(V,E)$, a schedule σ that satisfies all timing constraints can be computed as the SINGLE SOURCE LONGEST PATH lengths from $a \in V$. A positive cycle in the graph indicates a conflicting set of timing constraints that cannot be satisfied.

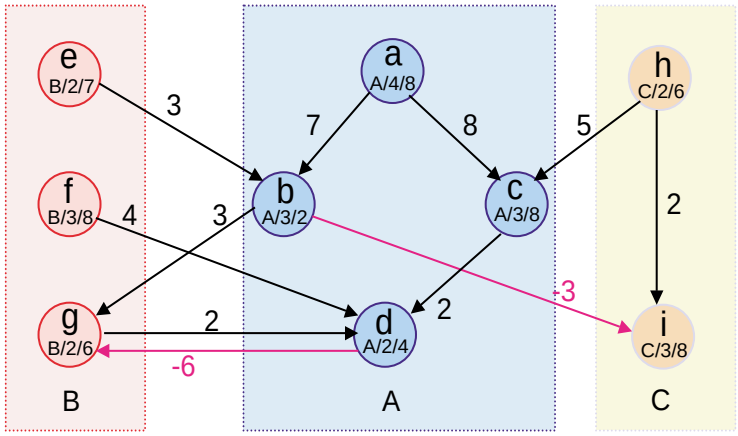


Figure 3.1: Constraint graph of a scheduling problem.

A schedule computed by Lemma 1 must satisfy all timing constraints. In addition, a feasible schedule must not have any resource conflict, that is, tasks that share the same resource must be serialized.

Definition 5 (Time-validity of a schedule) Given a scheduling problem with a task set T, a resource set R and a timing constraint set C, a schedule σ is *time-valid* if

(1) σ satisfies all timing constraints in C, and

(2) $\forall$ tasks $u, v \in T$ such that $r(u) = r(v) \in R$, u and v must be serialized, that is, either $\sigma(u) + d(u) \leq \sigma(v)$ or $\sigma(v) + d(v) \leq \sigma(u)$ holds.

Lemma 1 indicates the way to use graph algorithms to solve scheduling problems with timing constraint. The constraint graph can also be used to serialize tasks on shared resources, as required by Definition 5 to obtain a time-valid schedule. Serialization can be performed by adding extra edges to constraint graph G. For example, to serialize task v after u, an edge $(u, v) : d(u)$ can be added to G, such that $\sigma(u) + d(u) \leq \sigma(v)$ is guaranteed.

Given a time-valid schedule σ, there are alternative choices for start time assignment $\sigma(u)$ to a task u. We extract these available time slots as slacks of tasks. Slack is a measure of how much a task can be delayed without invalidating the schedule.

Definition 6 (Constraint slack Δ^c_σ) Given a time-valid schedule σ computed from a constraint graph $G(V,E)$,

(1) $\forall$ edge $(u,v) : \delta \in E$, the *edge slack* of edge (u,v) is defined as $\Delta^c_\sigma(u,v) = \sigma(v) - \sigma(u) - \delta$.

(2) For any task u represented by a vertex $u \in V$, the *constraint slack* of u is the minimum among all edge slacks of u's outgoing edges, that is, $\Delta^c_\sigma(u) = min(\Delta^c_\sigma(u,v))$, $\forall$ vertex v such that $(u,v) \in E$.

(3) If u does not have any outgoing edges, $\Delta^c_\sigma(u) = \tau_\sigma - \sigma(u) - d(u)$.

Lemma 2 If a schedule σ is time-valid, then a modified schedule σ' does not violate any timing constraints if it is identical to σ, except that the start time of task u is delayed until another time $\sigma'(u)$ within its constraint slack $\Delta^c_\sigma(u)$, that is $0 < \sigma'(u) - \sigma(u) \leq \Delta^c_\sigma(u)$, for a specific task u.

The constraint slack of a task u defines the maximum time unit by which it can be delayed without violating any timing constraint. It is calculated by the outgoing edges from u. If there is no outgoing edges, u can be delayed all the way until it completes at the finish time of the schedule. Such a delay will maintain the satisfaction of all timing constraints, but it may introduce new resource conflicts.

Definition 7 (Resource slack Δ^r_σ) Given a time-valid schedule σ computed from a constraint graph $G(V,E)$, for any task u represented by a vertex $u \in V$,

(1) If $\exists$ a task v that is mapped to resource $r(u)$ and v is scheduled after u, task u's *resource slack* is defined as $\Delta^r_\sigma(u) = min(\sigma(v)) - \sigma(u) - d(u)$, $\forall v$ such that $r(v) = r(u)$ and $\sigma(v) > \sigma(u)$.

(2) If such v does not exist, $\Delta^r_\sigma(u) = \tau_\sigma - \sigma(u) - d(u)$.

Lemma 3 If a schedule σ is time-valid, then a modified schedule σ' does not have any resource conflicts if it is identical to σ, except that the start time of task u is delayed until another time $\sigma'(u)$ within its resource slack $\Delta^r_\sigma(u)$, that is $0 < \sigma'(u) - \sigma(u) \leq \Delta^r_\sigma(u)$, for a specific task u.

The resource slack of a task u represents the vacant time slots between u's completion and the start of the next task on resource $r(u)$. If u is the last task scheduled on $r(u)$, then it can be delayed all the way until it completes at the finish time of the schedule. The new schedule remains time-valid if the delay on u does not exceed both its constraint slack and resource slack.

Definition 8 (Slack Δ_σ) Given a time-valid schedule σ computed from a constraint graph $G(V,E)$, for any task u represented by a vertex $u \in V$, its *slack* is defined as the minimum of its constraint slack and its resource slack, $\Delta_\sigma(u) = min(\Delta^c_\sigma(u), \Delta^r_\sigma(u))$.

Lemma 4 (Slack-bounded time-validity) If a schedule σ is time-valid, then a modified schedule σ' is also time-valid if it is identical to σ, except that the start time of task u is delayed until another time $\sigma'(u)$ within its slack $\Delta_\sigma(u)$, that is $0 < \sigma'(u) - \sigma(u) \leq \Delta_\sigma(u)$, for a specific task u.

Given a time-valid schedule, Lemma 4 allows some tasks to be delayed while yielding new schedules that are also time-valid. The slack properties of tasks form the basis of our power-aware scheduling algorithms for power/performance trade-offs.

3.4.2 Power characteristics of a schedule

We extend the power properties to schedules based on the constraint graph formulation. A schedule has a power profile representing the power consumption of task execution.

We introduce max and min power constraints and extract some new properties by applying power constraints to a schedule.

Definition 9 (Power profile P_σ, Total energy E_σ) Given a time-valid schedule σ,

(1) The *power profile* of σ is defined as a function of time. At any given time t, its value is the total power consumption of all tasks that are being executed at t. That is, $P_\sigma(t) = \sum p(u)$, $\forall$ task $u \in T$ such that $\sigma(u) \leq t \leq \sigma(u) + d(u)$.

(2) The *total energy* of σ is the integral of its power profile over time, that is, $E_\sigma = \int_0^{\tau_\sigma} P_\sigma(t)dt$.

Definition 10 (Max and min power constraints P_{max} and P_{min}) The power profile P_σ is constrained by two parameters, $P_{max}, P_{min} \in \mathbb{R}$, $P_{max} \geq P_{min} \geq 0$.

(1) The *max power constraint* P_{max} specifies the maximum level of supply power that can be provided to support task execution.

(2) The *min power constraint* P_{min} specifies the level of power consumption to maintain a preferred magnitude of activity.

We treat the max power constraint as a hard constraint. At any given moment, the total power consumption by all running tasks must not exceed P_{max}. The min power constraint is a soft constraint. The scheduler should make the best effort to meet the min power goal, in order to fully utilize free power such as solar, as well as to control the amount of jitter in power profile.

Definition 11 (Power spike, power gap) Given a schedule σ with its power profile $P_\sigma(t)$, and power constraints P_{max} and P_{min},

(1) At any given time t_1, if the power profile $P_\sigma(t_1)$ exceeds max power constraint, that is, $P_\sigma(t_1) > P_{max}$, then the power profile at time $t1$ is called a *power spike*.

(2) At any given time t_2, if the power profile $P_\sigma(t_2)$ is below min power level, that is, $P_\sigma(t_2) < P_{min}$, then the power profile at time $t2$ is called a *power gap*.

Power spikes and power gaps are the times slots where the power constraints are violated. Since only the max power constraint is treated as a hard constraint, a schedule with any power spikes must not be considered as a valid one. However, power gaps will not invalidate a schedule. Accordingly, a valid schedule is defined as follows.

Definition 12 (Power-validity of a schedule) Given a time-valid schedule σ computed from a constraint graph G with the task set T, constraint set C, and resource set R, for a max power constraint P_{max}, schedule σ is *power-valid* if,

(1) σ is time-valid by Definition 5, and

(2) Its power profile does not exceed max power constraint, that is, $P_\sigma(t) \leq P_{max}$, for $0 \leq t \leq \tau_\sigma$.

Definition 12 incorporates the power usage of a schedule as a constraint in addition to the existing constraints on the time dimension. Only max power constraint is used to qualify the validity of a schedule. (In the ensuing text, if not explicitly specified, a "valid" schedule means it is power-valid, which implies its time-validity.) Min power usage, which refers to the utilization to free power sources, is not enforced. Such separation distinguishes different power sources as expensive power and free power. It forms some new perspectives on power/performance trade-offs in a power-aware system, as described in the following definitions.

Definition 13 (Power cost $Pc_\sigma(P_{min})$, Energy cost $Ec_\sigma(P_{min})$) Given a time-valid schedule σ with a min power constraint P_{min} reprsenting free power level,

(1) The *power cost* of σ is the power usage above the min power level P_{min}. It is

defined as a function of time and P_{min},

$$Pc_\sigma(P_{min},t) = \left\{ \begin{array}{ll} P_\sigma(t) - P_{min} & \text{when } P_\sigma(t) > P_{min} \\ 0 & \text{when } P_\sigma(t) \leq P_{min} \end{array} \right\} \text{for } 0 \leq t \leq \tau_\sigma$$

(2) The *energy cost* is the integral of the power cost function,
$Ec_\sigma(P_{min}) = \int_0^{\tau_\sigma} Pc_\sigma(P_{min},t)dt$.

Definition 14 (Min power utilization $\rho_\sigma(P_{min})$) Given a time-valid schedule σ with a min power constraint $P_{min} > 0$ reprenting free power level, its *min power utilization* is defined as the ratio of its energy drawn from free power source over the total available free energy, $\rho_\sigma(P_{min}) = \frac{E_\sigma - Ec_\sigma(P_{min})}{P_{min} \times \tau_\sigma}$.

We do not limit the power and energy costs and min power utilization in Definitions 13 and 14 to only a power-valid schedule, since these properties are also meaningful to schedules that are not power-valid. They further highlight the difference between costly power and free power. Any power consumption below the min power level does not contribute to the energy consumption from non-renewable energy sources. In fact, the free power should be utilized greedily to preserve the costly power. This new perspective subsumes the conventional power or energy minimization techniques as a special case, where $P_{min} = 0$. A power-aware design should explore different trade-offs between *performance vs. costly power*, while making the best effort to fully utilize the free energy for performance speedup. This forms the basis of our power-aware scheduling techniques presented in Section 3.5.

3.4.3 Power-aware Gantt chart

There exist various visual representations for real-time scheduling problems, e.g. Gantt chart. However, very few of them have the capability to express power properties of a schedule, regardless of any power constraints. We introduce our *power-aware Gantt*

chart as a new visual representation for power-aware scheduling problems. It presents a schedule in two different views: *time view* and *power view*. Each view is a two dimensional diagram whose horizontal axis represents time and vertical axis represents power. In the time view, tasks are displayed as bins placed on several rows that denote parallel execution resources. The power view shows the power profile of the schedule with min and max power constraints and some corresponding power properties.

In the time view for a schedule σ computed from a constraint graph G with task set T and resource set R, the execution of a task $u \in T$ is represented by a horizontal bin beginning with its start time $\sigma(u)$ and whose length corresponds to its duration $d(u)$. We scale the vertical size of the bin to denote power consumption $p(u)$. As a result, the area of the bin indicates its energy expenditure. Each execution resource $R_i \in R$ takes one row denoted by R_i. All tasks that are mapped on this resource, that is, $\forall u \in T$ such that $r(u) = R_i$, are displayed in row R_i in timing order. The empty time slots between adjacent bins represent the resource slacks. Timing constraints, and slacks in the time dimension, though normally not shown, can also be intuitively visualized by selectively attaching annotation on the bins.

By collapsing all bins in the time view to the lowest horizontal axis, the expected power profile $P_\sigma(t)$ can be shown in the power view of the power-aware Gantt chart. It also illustrates the composition of the power profile from every power consumer's contribution at each time. With annotation of max and min power level, power spikes and power gaps can be directly observed; the power/energy cost vs. free power usage are clearly separated; and power properties such as power/energy cost, $Pc_\sigma(P_{min}, t), Ec_\sigma(P_{min})$ and min power utilization $\rho_\sigma(P_{min})$ can be visualized with the corresponding annotations.

Fig. 3.2 shows the power-aware Gantt chart of a time-valid schedule to the example problem in Fig. 3.1. In addition to a graphical representation to schedules, the power-aware Gantt chart also serves as the underlying model for a power-aware design tool

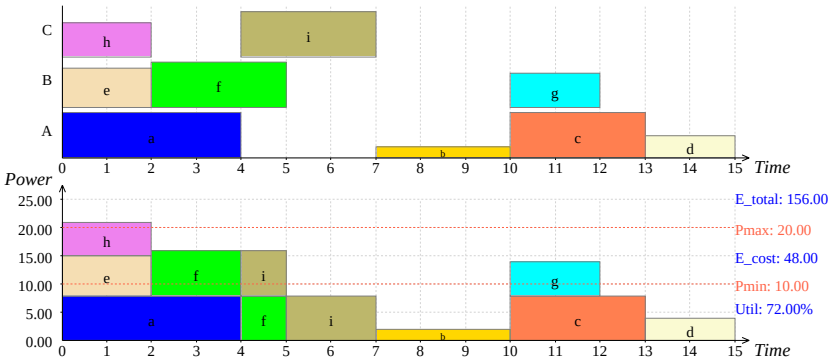


Figure 3.2: Power-aware Gantt chart of a time-valid schedule.

that allows the designers to evaluate different power/performance trade-offs visually. The designers can manually intervene with the automated scheduling process by dragging and locking the bins to alternative time slots in the time view, while observing the results in the power view interactively.

3.5 Algorithm

Based on the constraint graph formulation, we develop graph algorithms for power-aware scheduling. Given a scheduling problem, the goal of the power-aware scheduler is to find a valid schedule σ with following properties. (1) σ must be time-valid, that is, it satisfies all timing constraints and can arrange all tasks to corresponding execution resources without any resource conflicts. (2) σ must satisfy the max power constraint, that is, no power spikes can be found in the schedule. By qualifying (1) and (2) the schedule is a valid one that meets all hard constraints. (3) σ could have power gaps according to the min power constraint, but the scheduler should make its best efforts to remove power gaps, by reducing power/energy cost or improving min power utilization.

Power-aware scheduling is a multi-constraint solving problem. Our approach is to first examine different constraints in our model defined in Section 3.4. We find that

the constraints on timing and resource sharing are the most critical ones that must be considered first as necessary conditions. Next, we consider max power constraints after after a time-valid schedule is found. The scheduler must eliminate all power spikes while keeping the schedule time-valid to generate a valid schedule. Finally, the min power constraint can be applied after a valid schedule is given. The analysis suggests an incremental approach by solving one type of constraint at a time in the following three steps.

First, based on the constraint graph of the problem, we try to find a schedule that is time-valid. Power constraints and power consumption of tasks are not considered in this step. The algorithm is presented in Section 3.5.1. It extends previous work on a time-driven serial scheduling for a single execution resource to handling parallel execution on multiple resources.

Second, after a time-valid schedule is computed from the first step, the max power constraint is applied to constrain its power profile. Section 3.5.2 explains the algorithm to remove power spikes by using heuristics based on slack properties of the schedule. Tasks that contribute to a power spike are partially reordered by a slack-based ordering function. To avoid exhaustive search in the solution space, we apply heuristics to examine more reasonable solutions first.

Finally, given a valid schedule provided by the previous step, we apply the min power constraint and reorder tasks within their slacks to reduce power gaps and improve the min power utilization. The algorithm is illustrated in Section 3.5.3. It does not guarantee full utilization of the min power level. Also, the final schedule should not have a longer finish time with a loss of performance, since min power is a soft constraint that is not critical to the applicability of the schedule.

3.5.1 Algorithm for timing scheduling

The time-constrained scheduling algorithm is shown in Fig. 3.3. It is a extension to a previous serialization algorithm [18]. *G* is the constraint graph for the scheduling problem. *anchor* is the source vertex that is used in SINGLE SOURCE LONGEST PATH algorithm. It represents a virtual task that starts at time 0. *c* is called the candidate vertex that is being visited at each step as the algorithm traverses graph *G* topologically. The start time of candidate *c* is assigned as the distance from the *anchor* to *c* in the longest path. The next candidate *v* is selected from *c*'s successors. Tasks that share the same resources are serialized by adding edges between vertices. If these additional edges for serialization produce any positive loops in the graph, they are then removed by the algorithm and another topological ordering is attempted. The first invocation to the algorithm starts from *anchor* as the first candidate. Then the algorithm is recursively invoked at each step when a new candidate is selected. A time-valid schedule is returned when all vertices are scheduled.

This algorithm can be proved to always find a time-valid schedule if one exists, since it will traverse all possible topological orderings of the graph before it terminates with a failure.

Based on the problem shown in Fig. 3.1, its time-valid schedule is illustrated in Fig. 3.2 in the form of a power-aware Gantt chart. There are one power spike and several power gaps left for the remaining steps of our power-aware scheduler.

3.5.2 Algorithm for max power scheduling

The approach to meeting max power constraint is to eliminate the power spikes of a time-valid schedule computed by the previous step. The algorithm is shown in Fig. 3.4. It has three parameters: graph *G*, vertex *anchor*, and max power constraint P_{max}. The timing scheduler is always called first to obtain a time-valid schedule. The algorithm examines the power profile P_σ of the returned schedule σ to find the first power spike

```
TimingScheduler(Graph G, vertex anchor, vertex c)
    La := SINGLE SOURCE LONGEST PATH(G, anchor)
    if (positive cycle found) then
        return FAIL
    C := set of topological successors of candidate c
    if (C = ∅) then
        return σ with σ(c) := La
    while (C ≠ ∅) do
        v := one topological successor of C
        C := C − {v}
B:      foreach u ∈ C do
            if u ∉ v's successors
                then add u to v's successors
            if (r(c) = r(u)) then
                serialize u after c
        w := the most recently scheduled task, such that (r(w) = r(v))
        if (w ≠ nil) then
            serialize v after w
        σ = TimingScheduler(G, anchor, v)
        if (σ ≠ FAIL) then
            return σ with σ(c) := La
        undo added edges to G since step B
    return FAIL
```

Figure 3.3: Algorithm for timing scheduling.

```
MaxPowerScheduler(Graph G, vertex anchor, P_max)
    σ := TimingScheduler(G, anchor, anchor)
    if (σ = FAIL) then
        return FAIL
    for (t := 0; t ≤ τ_σ; t := t + 1) do
        S := set of all active tasks at t, ordered by slack Δ_σ
        power := P_σ(t)
        reschedule := FALSE
        while (power > P_max or reschedule = TRUE) do
B:          repeat
                v := EXTRACT MAX(S)
                if (reschedule = FALSE and Δ_σ(v) = 0) then
                    reorder tasks in S by constraint slack Δ^c_σ
                    reschedule := TRUE
                delay v by some time units (heuristically determined)
                power := power − p(v)
                S := S − {v}
            until (power ≤ P_max or S = ∅)
            if (S = ∅) then
                return FAIL
            if (reschedule = TRUE) then
                lock start time of all tasks in S
                σ := MaxPowerScheduler(G, anchor, P_max)
                if (σ ≠ FAIL) then
                    return σ
                undo added edges to G since step B
    return σ
```

Figure 3.4: Algorithm for max power scheduling.

at time t. To eliminate the spike, several simultaneous tasks at t are delayed so that the height of the power curve is less than P_{max}. The algorithm itself is called recursively after the spike at t is eliminated by delaying tasks. A valid schedule σ is found if there is no power spike in σ; and the time-validity of σ is always guaranteed. If no solution can be found after the recursive call, a failure notice is returned suggesting that either additional tasks at t need to be delayed, or one or more tasks already delayed have been incorrectly chosen.

The key issues in this algorithm are properly selecting and delaying tasks for spike

elimination. We do not attempt exhaustive enumeration to all possible partial orders of tasks which would take exponential orders of total number of tasks. Therefore, some heuristics must be applied. The badly chosen tasks could have several impacts. First, the total execution time τ_σ may be extended unnecessarily, leading to a loss of performance. Second, the algorithm may evaluate some invalid schedules repeatedly before approaching a valid one, so that the scheduler requires extra computation time needlessly. Finally, the algorithm may fail to find a valid schedule even if one exists.

We propose slack-based heuristics for selecting and delaying tasks. First, a slack-based heuristic function is used to order simultaneous tasks. When a power spike is detected at time t, the algorithm orders tasks that are active at t by their slacks Δ_σ(Definition 8), and then selects tasks to delay based on the following conditions. (1) If there are tasks with non-zero slacks, the task with the largest slack is always selected first. The algorithm continues selecting tasks to delay until the power spike at t is removed. (2) If no tasks with non-zero slack is available while the power spike at t is still present, the remaining tasks are reordered by their constraint slacks Δ^c_σ(Definition 6). Tasks with larger constraint slacks will be delayed. (3) If the power spike cannot be removed until all the remaining tasks have zero constraint slack, tasks are randomly selected to be delayed.

After a task is selected to be delayed, the second question is by how long it should be delayed, which is referred to as the *delay distance*. To delay a task u based on an existing schedule σ, we add an edge from *anchor* to u, with positive weight t' as the lower bound on its new start time. Therefore, the delay distance is $t' - \sigma(u)$. Clearly, making a small delay distance is not efficient. On the other hand, we do not expect the delay distance to be too large such that the finish time of the schedule may be unnecessarily increased. We currently heuristically set the upper bound of the delay distance to the execution time of the task. In addition, in case (1) where the selected task u has some slack, the delay distance is further bounded by its slack $\Delta_\sigma(u)$. According to the

slack-bounded time-validity (Lemma 4), if the delay distance of u is less than its slack, the new schedule is still time-valid. Therefore, in case (1), we put this extra bound to reduce the effort for rescheduling for time-validity. The algorithm can still proceed with a time-valid schedule. While in cases (2) and (3), since the new schedule after the delay is no longer time-valid, the timing scheduler must be invoked to make the schedule time-valid again by asserting the Boolean variable *reschedule* . In case (2), the selected task u has some constraint slack but no resource slack, the delay distance is further bounded by its constraint slack $\Delta^c_\sigma(u)$, so that all timing constraints are preserved thus the scheduler only needs to eliminate the resource conflict caused by the delay. All of these constraints can actually serve the purpose of pruning out the search space tremendously. Finally, in case (3), which eliminates a power spike at the cost of introducing new timing violations, some significant timing adjustment to the schedule is expected.

After enough tasks are delayed and the power spike at t disappears, we lock the start time of the remaining tasks. The start time of a task u is locked by adding two edges to graph G, a forward edge $(anchor, u) : \sigma(u)$, and a backward edge $(u, anchor) : -\sigma(u)$. As a result, task u is forced to start at time $\sigma(u)$ by the SINGLE SOURCE LONGEST PATH algorithm. These locks are especially meaningful to case (3). When the scheduler delays a task u to eliminate a power spike at time t, it is desirable to keep all tasks that are scheduled before t intact. While in case (3), if the delayed task u has an outgoing backward edge (u, v) such that task v is scheduled before t, the delay to u will force v to be also delayed. In fact, an attempt to remove a power spike starting at time t by delaying a task u may cause a new power spike before t. Such a result will certainly complicate the scheduler. The algorithm could spend much more time dealing with the unexpected spikes before it converges to a valid schedule. By locking the tasks that do not form a power spike at t, no further delays can be applied to these tasks. However, if delays to these tasks are necessary for a valid schedule, the algorithm will fail in its

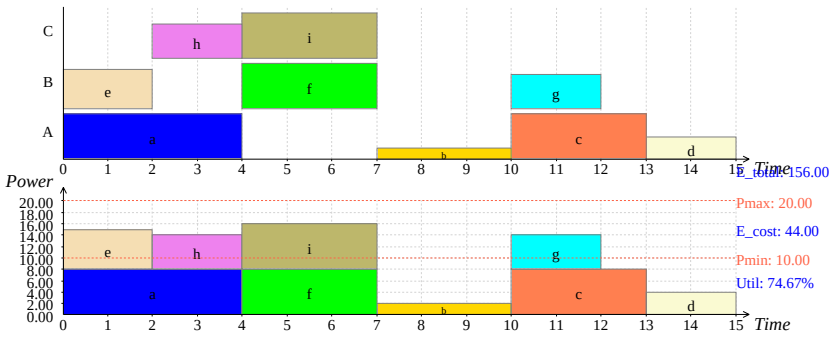


Figure 3.5: A valid schedule after max power scheduling.

next recursions and these locks will be undone. Then the algorithm will choose one task from them to make further delay and continue recursion.

It is notable that in some extreme cases, the max power constraint scheduler may not be able to find a valid schedule even though one exists . The reason is that the algorithm does not enumerate all possible combinations in partially ordered tasks. However, in practice, our heuristics perform very well in finding a valid solution without sacrificing performance. Our slack-based heuristics tend to examine more reasonable schedules first. Also, the heuristic to lock the tasks before the recursion can help reduce the computation of the scheduler.

The schedule shown in Fig. 3.2 does not satisfy the max power constraint. Fig. 3.5 show the valid schedule after applying the max power scheduler. Tasks *h* and *f* are delayed to remove the power spike.

3.5.3 Algorithm for min power scheduling

The goal of the min power constraint scheduler is to reduce the energy cost by improving min power utilization for a given valid schedule. The algorithm is shown in Fig. 3.6. Four parameters are passed to the algorithm: graph *G*, vertex *anchor*, power constraints P_{max} and P_{min}. A valid schedule σ is obtained from the power-valid scheduler at the beginning of the algorithm. If σ already has full min power utilization,

```
MinPowerScheduler(Graph G, vertex anchor, P_max, P_min)
    σ := MaxPowerScheduler(G, anchor, P_max)
    if (σ = FAIL) then
        return FAIL
    if (ρ_σ(P_min) = 1) then
        return σ
    improvement := TRUE
    while (improvement = TRUE) do
        improvement := FALSE
        for (t in a heuristic order of range (0, τ_σ)) do
            if (P_σ(t) < P_min) then
                S := set of tasks that start before t
                foreach u ∈ S such that Δ_σ(u) ≥ t − σ(u) − d(u) do
                    σ' := σ
B:                  delay u some time units such that u is active at t
                    if (σ is valid and ρ_σ(P_min) > ρ_σ'(P_min)) then
                        improvement := TRUE
                        if (ρ_σ(P_min) = 1) then
                            return σ
                    else
                        undo added edges to G in step B
                        σ := σ'
    return σ
```

Figure 3.6: Algorithm for min power scheduling.

then no further improvement is necessary, and the algorithm completes. Otherwise, it tries to find a power gap at time t to and delay some tasks scheduled before t to fill this power gap. These tasks must have enough slacks to be delayed until t such that the new schedule is time-valid. The algorithm also checks whether the new schedule has any power spikes, and whether its min power utilization is better than the existing schedule. If so, it is a better schedule and the algorithm continues searching for further improvement. Otherwise, the delay is cancelled and the previous schedule is restored.

In order to find an "optimal" schedule whose energy cost is the minimized, the algorithm should examine all valid partial orderings of tasks, which will increase the complexity of computation to an exponential order of tasks. Therefore, we apply heuristics based on following observations. First, the scheduler may need to scan the schedule multiple times. This is because delaying tasks to fill a power gap at time t may create new power gaps before t. Also, since delaying one task u will change the slacks of other tasks that are constrained by u, there may be new opportunities for reordering those tasks that are not eligible for delay previously. As a result, either new power gaps or new tasks to fill other power gaps can be found after the algorithm scans the schedule again. Moreover, the order in which to visit the power gaps will lead to different final schedules because different partial reorderings of tasks are applied. This suggests that better schedules could be found if we scan the schedule in various orders in time dimension, e.g. incremental order, reverse order, or random order. Finally, when a task u is selected to fill a power gap at t, we consider alternative time slots to reschedule u, rather than just starting u at t. It is difficult to determine the "best" time slot for rescheduling u since it alters not only the power profile but also the slacks of some other tasks. We also address this issue by heuristics. Some available heuristics are: starting u at t, finishing u at the end of the power gap starting from t, or a randomly chosen time slot. In practice, we can scan the schedule multiple times while altering some of the heuristics during each scan and take the best results.

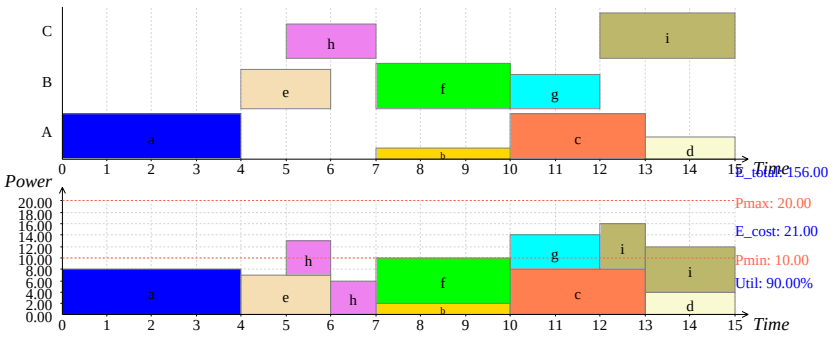


Figure 3.7: The improved schedule after min power scheduling.

The Boolean variable *improvement* refers to whether the scheduler finds a better schedule during one scan to the existing schedule. If no further delay can improve the schedule, the algorithm terminates successfully. Each scan (except the last one) will improve the schedule by delivering the same performance with a reduced energy cost. Since min power constraint is a soft constraint, the schedule tolerates the existence of power gaps after it makes the best efforts to remove them.

In this algorithm, tasks are delayed within their slacks during schedule improvement. This guarantees that the delays always result in time-valid schedules. The algorithm also never introduces delays that either create power spikes or incur a higher energy cost. Therefore, no additional rescheduling will be necessary after delaying these tasks. Furthermore, the min power scheduler can possibly reduce the height of the power profile. This indicates the same schedule can be applied to different power constraints without any extra effort to reschedule the problem.

Fig. 3.7 shows a better schedule that improves on the valid schedule in Fig. 3.5. Energy cost is reduced while the height of the power profile curve is also reduced. In fact, the same schedule can be directly applied to all cases with a range of constraints where $15 \leq P_{max} \leq 20, 2 \leq P_{min} \leq 15$, without recomputing a schedule for each case. This feature makes our statically computed power-aware schedules directly adaptable to a run-time scheduler that schedules tasks according to the dynamically changing

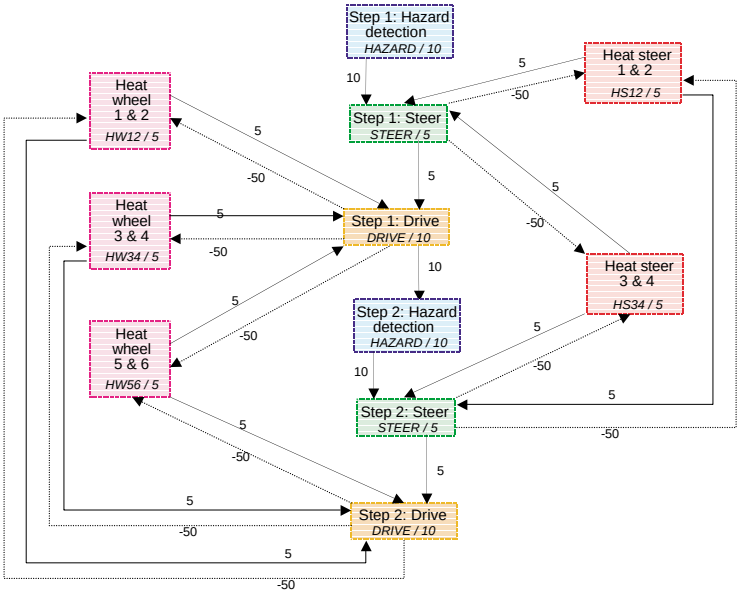


Figure 3.8: Constraint graph of the Mars rover.

constraints imposed by the environment.

3.6 Experimental Results

This section presents scheduling results for the Mars rover operations and a case study for evaluating our power-aware scheduling algorithms in a mission scenario.

The constraint graph for the Mars rover is shown in Fig. 3.8. Since the power consumption varies in three different cases, the power attributes of tasks are not shown in the graph. To simplify the problem, we assume all heaters are independent resources and one heater can heat two motors at a time. Therefore there are a total of five thermal heaters. Four steering motors are considered a single steering mechanical resource. The six wheel motors are modeled as one mechanical unit for driving. There is also a laser guided digital component for hazard detection.

Fig. 3.9, 3.10 and 3.11 show the results for three cases after applying power-aware

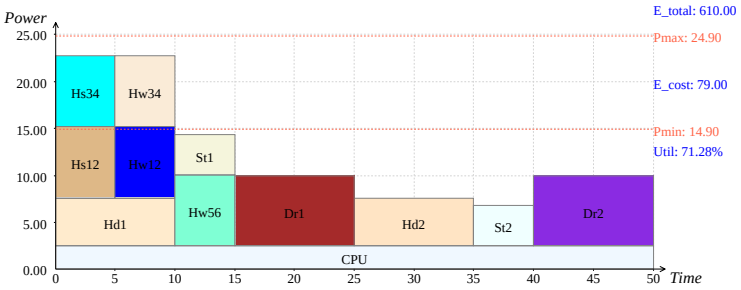


Figure 3.9: Schedule for the best case.

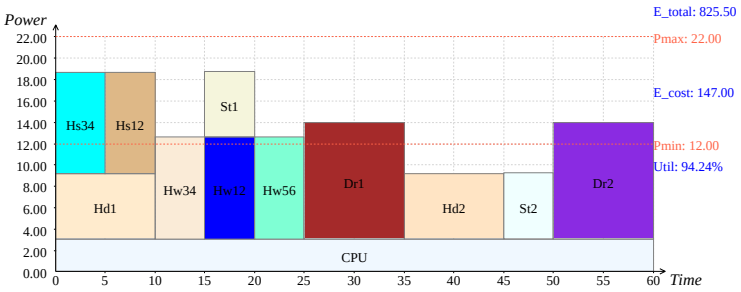


Figure 3.10: Schedule for the typical case.

scheduling algorithms. Fig. 3.9 gives first two iterations of the loop in the best case. To utilize the available free energy, we manually unroll the loop and insert two heating tasks to improve loop efficiency through better solar energy utilization. Therefore the second iteration can be repeated without too much energy cost. In other cases only one iteration is shown since loop unrolling is not necessary. In the best case, because the power budget is sufficient, a fast schedule is given by allowing operations to overlap. In the typical case, parallel operations are still possible while some heating tasks are serialized. In the worst case, a tight power budget forces all operations to be serialized, leading to a slow schedule.

The existing schedule used in the past mission was designed to be low-power. To avoid exceeding max power supply, JPL uses a serialized schedule that is fixed in all situations, regardless of available solar power and power consumption in different conditions. The existing schedule is identical to our power-aware schedule computed with

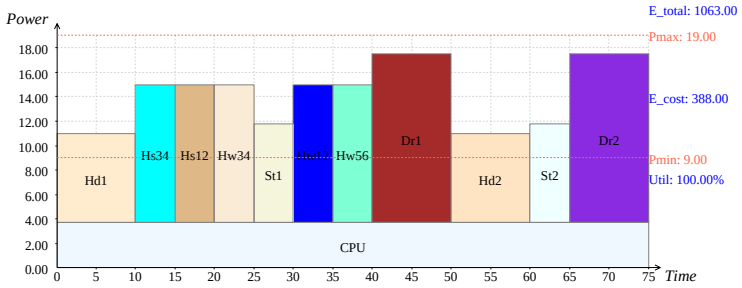


Figure 3.11: Schedule for the worst case.

Solar power (W)	Battery energy (J)	Solar energy (J)	% of solar energy	Time (s)	Moving distance
14.9	0	672.5	60%	75	2 steps - 14cm
12	55	817	91%	75	2 steps - 14cm
9	388	675	100%	75	2 steps - 14cm

Table 3.3: Performance of the rover under existing schedule.

the lowest min and max power constraints. The fundamental difference is that, our schedule is completely constraint-driven; whereas the existing solution is hardwired and does not track the power availability. The performance and energy cost of our schedules and the existing schedule are compared in Table 3.3 and Table 3.4.

We use finish time τ_σ and energy cost $Ec_\sigma(P_{min})$ to the non-rechargeable battery as the metrics. The existing scheme only schedules for the worst case; while in other cases, solar energy is under-utilized and opportunities to performance improvement are overlooked. However, JPL's low-power schedule appears "economic" since its energy cost is low. Our schedules, on the other hand, speeds up the rover's movement by up to 50% in the best case and 25% in the typical case, while drawing more costly energy from the battery. To evaluate this trade-off, we apply our schedules and the existing

Solar power (W)	Battery energy (J)	Solar energy (J)	% of solar energy	Time (s)	Moving distance
14.9	79.5 / 6	534	70%	50	2 steps - 14cm
12	147	679	94%	60	2 steps - 14cm
9	388	675	100%	75	2 steps - 14cm

Table 3.4: Performance of the rover under power-aware schedules.

Time frame (s)	Solar power (W)	JPL			Power-aware		
		Travel distance	Time (s)	Energy cost (J)	Travel distance	Time (s)	Energy cost (J)
0-599	14.9	16	600	0	24	600	145.5
600-1199	12	16	600	440	20	600	1470
1200 -	9	16	600	3114	4	160	776
Total		48	1800	3554	48	1360	2391.5
					Improvement	24.4%	32.7%

Table 3.5: Comparison of existing schedule to power-aware schedules under a mission scenario.

schedule to a mission scenario when the available solar power varies over time, and then evaluate the performance vs. energy cost in this bigger picture.

Suppose the mission is to travel to the next target location, which is 48 steps away from the current location. The mission starts around noon when maximum solar power is present. While the mission is in progress, the power output from the solar panel drops from 14.9W to 12W after 10 minutes, then falls to the worst case at 9W 10 minutes later. If the existing schedule is applied, the rover will spend 10 minutes evenly in the best case, typical case, and worst case since it has a fixed moving speed (16 steps per 10 minutes). This results in a long execution time (30 minutes) and considerable energy cost in the worst case. When our schedules are used, the rover finishes 50% of its work (24 steps) in the first 10 minutes, 42% of work (20 steps) in the next 10 minutes, leaving the remaining 8% (4 steps) in the worst case for less than 3 minutes. Since our schedules accelerate execution at the best and typical cases, the rover can finish the mission earlier before having to work in the costly worst case. The results of this case study are shown in Table 3.5. The analysis shows our schedules win both on performance and energy savings considerably.

Fig. 3.12 highlights the property of the power-aware scheduler in a geometrical view. The top chart illustrates how the power-aware scheduler adjusts the execution speed adaptively with available power budget, while the existing scheme ignores the power constraint and always operates at the lowest speed. The workload is represented by the integral of the speed curve over time. Therefore our curve reaches the given

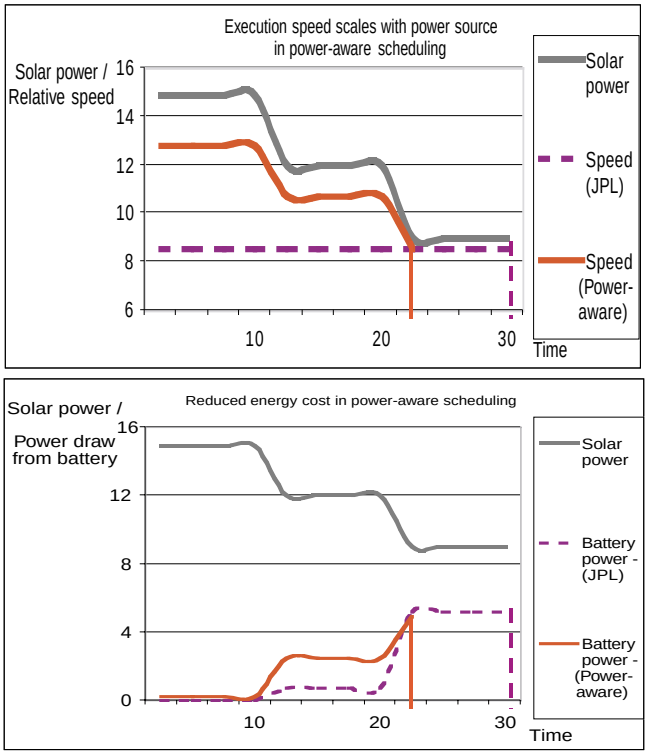


Figure 3.12: Adaptive speedup in power-aware scheduling.

workload earlier because of higher execution speed before operating in the worst case. The bottom chart shows the power cost from battery over time and how it alters as power constraint varies. The energy expenditure is symbolized by the integral of power curve over time. When the mission is completed, both the speed curve and power curve also end. Although our power curve is higher in most time during the mission, by completing earlier we avoid further energy cost from integrating a high power curve with a longer execution time. Therefore, given the same workload, the power-aware scheduler is capable of achieving performance speedup less energy cost simultaneously.

3.7 Chapter Summary

Power-aware design becomes a more important issue in mission-critical systems that require best use of available power sources and deliver high performance at the same time. We target the scheduling algorithms to embedded systems with variable power constraints and various types of power consumers, as well as different energy sources that are classified as costly power vs. free power. In these systems, power-aware techniques have potentials for both performance improvement and energy savings.

In this chapter, we present a constraint-driven model that incorporates power and timing constraints in a system-level context. We propose three core algorithms that decompose the power-aware scheduling problems into steps. Via this incremental approach, we distinguish the properties of each sub-problem and apply heuristics to solve the constraints by different methods. The case study to a real application demonstrates that our power-aware method is capable of improving performance while saving expensive energy.

Several interesting issues in this dimension need further attention. To expand the applicability of our algorithms, more effective heuristics need to be discovered. We would also like to incorporate more novel power management techniques including voltage/frequency scaling into this tool to support more effective power-aware designs.

Chapter 4

Power Aware Task Motion

New embedded systems are being built with new types of energy sources, including solar panels and energy scavenging devices, in order to maximize their utility when battery and A/C power are unavailable. The large dynamic range of these unsteady energy sources is giving rise to a new class of *power-aware* systems. They are similar to *low-power* systems when energy is scarce; but when energy is abundant, they must be able to deliver high performance and fully exploit the available power. To achieve the wide dynamic range of power/performance trade-offs, we propose a new *task motion* technique, which tunes the system-level parallelism to the power/timing constraints as an effective way to optimize power utility. Results on real-life examples show an energy reduction of 24% with a 49% speedup over best previous results on the entire system.

4.1 Introduction

Recent years have seen the emergence of *power-aware* embedded systems. They are characterized by not only low power consumption, but more generally by their ability to support a wide range of power/performance trade-offs. That is, these systems can

be viewed as providing “knobs” that can be turned one direction to reduce power consumption, or the other direction to increase performance. The ability to adapt the range of power/performance trade-offs is driven by new applications that demand very high performance while under stringent timing and power constraints.

One example that fits this description is the Mars rover by NASA/JPL [4]. It was designed to roam on Mars to take digital photographs and perform scientific experiments over several hundred days. Its energy sources consist of a battery pack and a solar panel, and future versions are expected to incorporate nuclear generators, thermal batteries, and energy scavenging devices. Besides the Mars rover, many new emerging embedded systems are also following this trend towards new types of heterogeneous, renewable energy sources. Future personal digital assistants (PDAs) will likely include solar panels as found in many calculators today. Yet another example is the distributed sensors. They are being built today to draw energy from solar power, wind power, or even ocean waves. They represent a great improvement because they enable the system’s continued operation for useful or critical tasks when the traditional energy sources like battery and A/C become unavailable.

These new types of energy sources are posing new challenges to designers of power-aware systems. What they all have in common is that many of these new energy sources are far from being ideal power supplies. For example, the output of a portable solar panel today can be up to 15W under direct sunlight, or down to 1mW under incandescent light. Similarly, other sources will be determined by the wind or ocean wave, which can also cause the available power to vary by several orders of magnitude. Embedded systems powered by such sources must be designed to operate in as wide a range as possible. Indeed, new emerging components such as the Intel XScale are able to scale their power/performance over $20\times$, and this dynamic range will likely to increase.

While low power operation is clearly important, the ability to fully exploit the avail-

able power when energy is abundant is equally important. However, today's systems let much free energy go to waste, because they are designed for fixed budgets. For example, a system with an XScale draws approximately 1W of power, but when the solar panel outputs 15W in direct sunlight, up to 1400% of the power will be wasted. Even if there is a rechargeable battery, when it becomes fully charged, the extra power turns into waste heat. This is also the case with the Mars rover, which accomplishes its low-power property by serializing all tasks, including mechanical and heating as well as computation. However, it also discards excess power as waste heat.

One way to take advantage of the excess power is to increase parallelism. In fact, parallelism is in general an effective way for both high performance and low power. By operating additional processors at their peak rate, they will be able to take advantage of the abundant energy. Parallelism can also enable a set of processors to operate at a lower power level than a single processor with the same performance. Although it is difficult to parallelize algorithms in general, systems with many concurrent activities present many opportunities for parallelism-based trade-offs.

Peak-power poses new challenges to such a power-aware architecture with multiple processors. Today's systems satisfy the peak-power constraint by construction, that is, each component is given a budget that is guaranteed never to be exceeded according to their data sheet. However, by using multiple processors to fully utilize the available power when abundant, a multiple processor architecture would risk exceeding the total budget when the supply power is low, if it is not designed carefully. Therefore, it is of utmost importance that the proposed scheme be able to fully respect the maximum power as a hard constraint.

In this chapter, we propose to enhance the dynamic range of these embedded systems by means of *task motion* and power-aware scheduling. It transforms tasks within their timing constraints and their precedence dependency in order to match the parallelism to the available power level. Furthermore, we exploit domain-specific knowl-

edge about the power-consuming tasks to achieve additional significant power/performance improvements over existing schedulers. The enhanced dynamic range and power-awareness enable the system to accomplish more tasks in a shorter amount of time while respecting all timing constraints. The benefits must ultimately be translated into application-specific metrics, but as power-aware systems are deployed in more mission-critical applications, the saving from reduced mission time or enhanced quality may translate into a saving of millions of dollars.

Section 4.2 reviews related work. Section 4.3 uses an example showing a counter-intuitive result when some of the well-known techniques will fail at the system level. However, this problem can be successfully addressed by our new technique, which is presented in Section 4.4. We discuss experimental results in Section 4.6.

4.2 Related Work

To explore the power/performance range in power-aware embedded systems, we can draw from many techniques developed for low power and high performance. This section surveys related works in these areas with a discussion on their integration at the system level.

Low power can be achieved by many ways. For system-level designs, where the components are largely off-the-shelf or already designed, the applicable techniques include subsystem shutdown and dynamic voltage scaling (DVS). In the first case, subsystem shutdown decision can be based on fixed idle times, adaptive timeout, or predictive based on a mix of profile and runtime history [64, 62, 20]. Similarly, power-up may be either event-driven or predictive in an attempt to minimize timing or power penalty. In the second case, DVS techniques have been developed for variable-voltage processors (introduced by [74], with follow-up by [26, 48] and more). Because energy is a quadratic function of voltage, lowering the voltage can result in significant saving while still enabling the processor to continue making progress, unlike the shutdown

case. Lowering the voltage will also require reduction in frequency, which has the effect of reducing dynamic switching power.

In addition to low power, the power/performance range can also be increased towards high performance by drawing from previous works on retiming or pipelining and applying them to the system level. Leiserson et al. first established the theoretical foundation for retiming synchronous circuits [42], and this has been extended to loop scheduling for VLIW processors [56, 15, 35]. Shifting tasks in a data flow graph (DFG) across the iteration boundary can result in a shorter execution time or alleviate the resource pressure (e.g. number of registers and functional units). Such techniques are also used in power minimization by reducing switching activities [41, 75].

Existing techniques need significant enhancements before they can be correctly applied to a system-level power management problem. First, most techniques to date treat either power or timing as an *objective*, rather than a *constraint*. In real systems, the max power budget is a real, hard constraint, whose violation can lead to malfunction. Max power was not of central concern previously, but as we consider additional power sources such as solar whose power output can vary, max power constraints must be strictly enforced. This becomes especially important as we increase the range of power and performance trade-offs by tuning the parallelism. Second, the tasks to be scheduled are related to each other not only by precedence, data dependency or deadline, but also related across different components by dependencies like *co-activation*, which must be correctly modeled for system-level power management, or else anomalies can occur. Co-activation means the execution of one task requires the power consumption of other dependent services or tasks. A simple example is that when the CPU is running, it imposes a co-activation dependency on the memory. Techniques such as DVS are designed mainly for minimizing CPU power, but they have not considered other components that have dependencies on the CPU. In fact, energy saved on the CPU may be more than offset by the increased energy consumed by the rest of the system.

The following section presents a simple example to illustrate such an anomaly with applying DVS without system-level considerations.

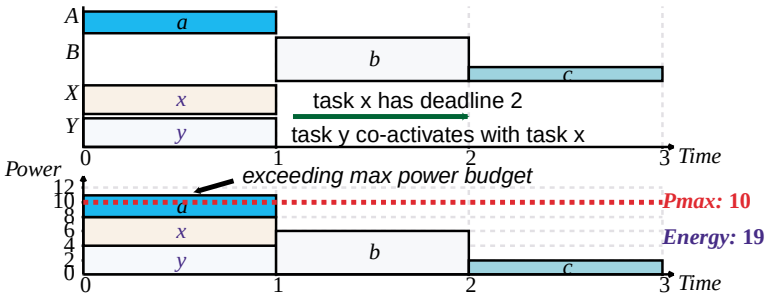
4.3 DVS Anomaly

We present a simple example in Fig. 4.1 to illustrate an anomaly with applying DVS without considering system-level dependencies, resulting in a suboptimal and incorrect system. It will be further used to explain our new system model and scheduling technique in the ensuing text. In this example, five tasks a, b, c, x, y are to be scheduled on four execution resources A, B, X, Y. The constraints are:

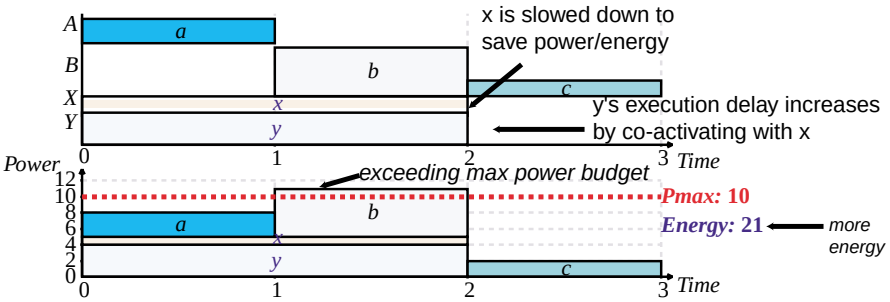
1. The overall deadline is at time 3.
2. The max power budget is 10W.
3. Tasks a, b and c must be serialized.
4. The execution resources A, B are not voltage-scalable.
5. Only task x can be voltage-scaled on resource X (e.g. a processor), and it has some slack time to finish before time 2.
6. Task y must be co-activate with task x, and its resource Y is also not voltage-scalable (e.g. memory, I/O).

Note that task y need not start and finish at the same time as x, but it must *envelop* x, i.e., start no later than x starts and finish no sooner than x finishes. For simplicity, this example assumes x and y start and finish together.

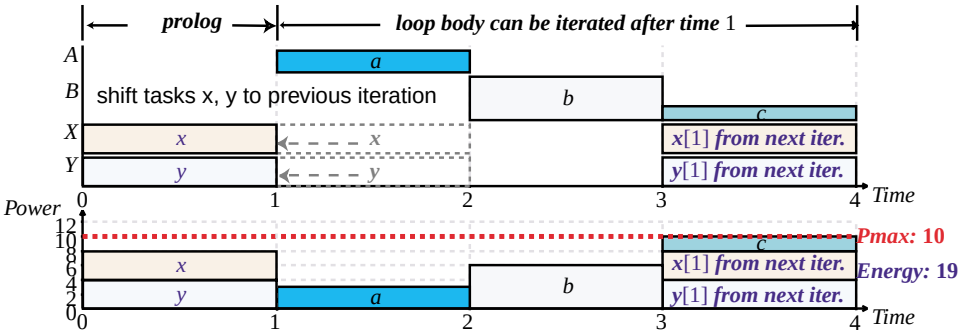
We present schedules as power-aware Gantt charts, where the horizontal and vertical axes represent time and power, respectively. Each chart also consists of a pair of views: *time view* organizes tasks by horizontal tracks that correspond to power consuming resources (processors, peripherals), and *power view* stacks the tasks over time



(a) The schedule is not valid since max power budget is exceeded at time slot [0,1] due to parallel tasks x, y and a.



(b) DVS technique reduces power and energy consumption of task x. However, it fails to produce a valid schedule to the entire system. The energy comsumption of the whole system is increased by co-activation.



(c) Our task motion technique shifts task x and its co-activated task y to the previous iteration such that the max power budget is satisfied.

Figure 4.1: An example where DVS fails to reduce power and energy at system level, while our new technique will succeed.

to show the power breakdown by tasks. The curve that traces the height of the power view is the *power profile* for the entire system.

Fig. 4.1(a) shows a time-valid schedule with a max-power violation during time $[0,1]$. Rescheduling x and y in $[1,2]$ will be time-valid but still violates max power. Fig. 4.1(b) shows the case when DVS was used to slow down task x until its deadline of time 2. Intuitively, reducing both power and energy of task x should eliminate the max power violation, but instead it not only does not reduce max power, but actually increases total energy at the system level. Because DVS slows down the processor, now the execution of x overlaps with task b, thereby leading to higher system-level power. Furthermore, because x runs more slowly, its co-activated task y must also consume power for longer but on a device that is not voltage scalable. Thus, energy saved by slowing down x is more than offset by the additional energy consumed by the lengthened y. This anomaly is an example where DVS should not be applied in isolation.

Fig. 4.1(c) shows a feasible solution obtained by our new *power-aware task motion* technique on iterative tasks. Task x and y are shifted (or *promoted*) to the previous iteration to overlap task c instead of a or b. As a result, both the max power and the deadline are satisfied. However, the optimal solution cannot be obtained unless we exploit domain-specific knowledge about the task set by eliminating a precedence dependency and replacing it with a *utilization constraint*. The details will be explained in later sections.

4.4 Task Motion under Timing and Power Constraints

We present a new power-aware task motion technique for evaluating power/performance trade-offs in embedded systems. We first define our constraint model and introduce our representations: a timing constraint graph G for scheduling, and the *iteration graph* G' for task motion. We also define *utilization constraints* to support more aggressive but

provably correct design space exploration.

4.4.1 Constraint graph and schedule

The input to the scheduler is a *(timing) constraint graph* $G(V,E)$, where the vertices V represent tasks, and the edges $E \subseteq V \times V$ represent timing constraints between tasks. Each vertex $v \in V$ has three attributes, $d(v)$, $p(v)$ and $r(v)$, representing task v's *execution delay*, *power consumption* and *resource mapping* respectively. Each edge $(u,v) \in E$ has two attributes, $\delta(u,v)$ and $\lambda(u,v)$. $\delta(u,v)$ specifies the *min/max timing constraints* [18]. For any function σ that assigns the start times to tasks u and v as $\sigma(u)$ and $\sigma(v)$, $\sigma(v) - \sigma(u) \geq \delta(u,v)$. If $\delta(u,v) \geq 0$, edge (u,v) is called a *forward edge* that specifies a *min timing constraint*. If $\delta(u,v) < 0$, it is a *backward edge* indicating a *max timing constraint*. $\lambda(u,v)$ is called the *dependency depth*, which specifies constraints across iterations. An *iteration* is a full pass of executing of each of the tasks once in a valid order. $\delta(u,v)$ and $\lambda(u,v)$ indicate that the execution of task u in iteration i must precede task v in iteration $i + \lambda(u,v)$ by $\delta(u,v)$ time units. If $\lambda(u,v) = 0$, edge (u,v) specifies an *intra-iteration constraint*. Otherwise, it is an *inter-iteration constraint*. We assume that inter-iteration constraints are only precedence dependencies (forward edges) and their dependency depths are positive integers. For backward edges, their dependency depths are always zero.

A *schedule* σ assigns a start time $\sigma(v)$ to each task $v \in V$. It has a *finish time* τ_σ when all tasks complete their execution. Schedule σ is called *time-valid* if all the start time assignments satisfy all timing constraints, and tasks that share the same resource are serialized. If G represents an iteration of a loop, σ must also satisfy inter-iteration constraints such that they must hold across iterations when multiple instances of σ are concatenated.

A schedule σ has a *power profile* function of time $P_\sigma(t), 0 \leq t \leq \tau_\sigma$ representing the instantaneous power consumption of all tasks during the execution of σ (illustrated by

the power view of the Gantt-chart in Fig. 4.1). The power profile is constrained by two parameters: P_{max}, P_{min}, such that $P_{max} \geq P_\sigma(t) \geq P_{min} \geq 0$. The *max power* constraint P_{max} specifies the maximum budget of supply power that can be provided by the power sources. The *min power* constraint P_{min} specifies the level of power consumption to maintain a preferred level of activity.

The max power constraint is a hard constraint. At any given time t, the value of the power profile function $P_\sigma(t)$ must not exceed P_{max}. Schedule σ is called *power-valid* (or simply, *valid*) if it is time-valid and its power profile does not exceed the max power constraint. However, we treat the min power constraint as a soft constraint that could be violated occasionally in a valid schedule.

In cases where the min power constraint P_{min} represents the free power level (e.g. solar), the energy drawn from the non-renewable energy sources is defined as the *energy cost* $Ec_\sigma(P_{min})$ of a schedule σ. It distinguishes between costly power and free power in such a way that any power consumption below the free power level does not contribute to the energy cost on non-renewable energy sources, and therefore should be utilized maximally.

4.4.2 Task motion under timing constraints

Task motion obtains different versions of a scheduling problem by converting between intra-iteration and inter-iteration constraints. We first construct an *iteration graph* $G'(V,E')$: it has the same vertices as those of the constraint graph $G(V,E)$, but edges E' consist of only intra-iteration constraints. Formally, $E' = \{(u,v) : (u,v) \in E$ such that $\lambda(u,v) = 0, \delta'(u,v) = \delta(u,v)\}$. The edges in E' will not have dependency depths λ, since they are always zero. The expected loop duration τ is obtained from the original schedule computed from the initial iteration graph G'.

Our work differs from previous works in several ways. First, existing techniques either do not consider timing constraints δ in their data flow graphs (DFG), or the value of

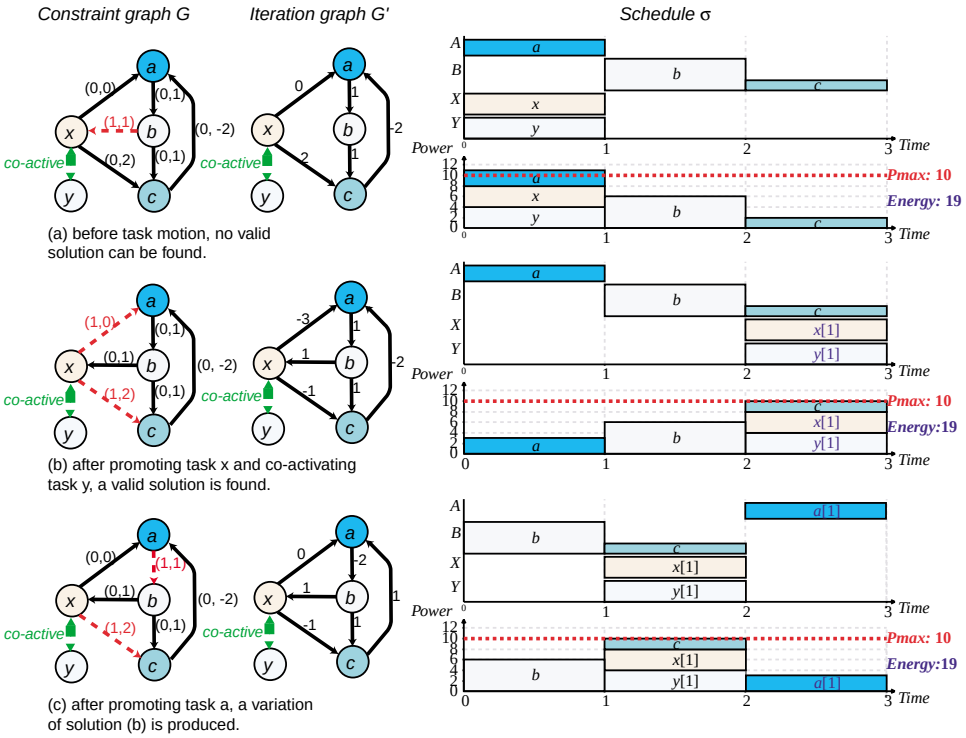


Figure 4.2: Task motion under timing constraints.

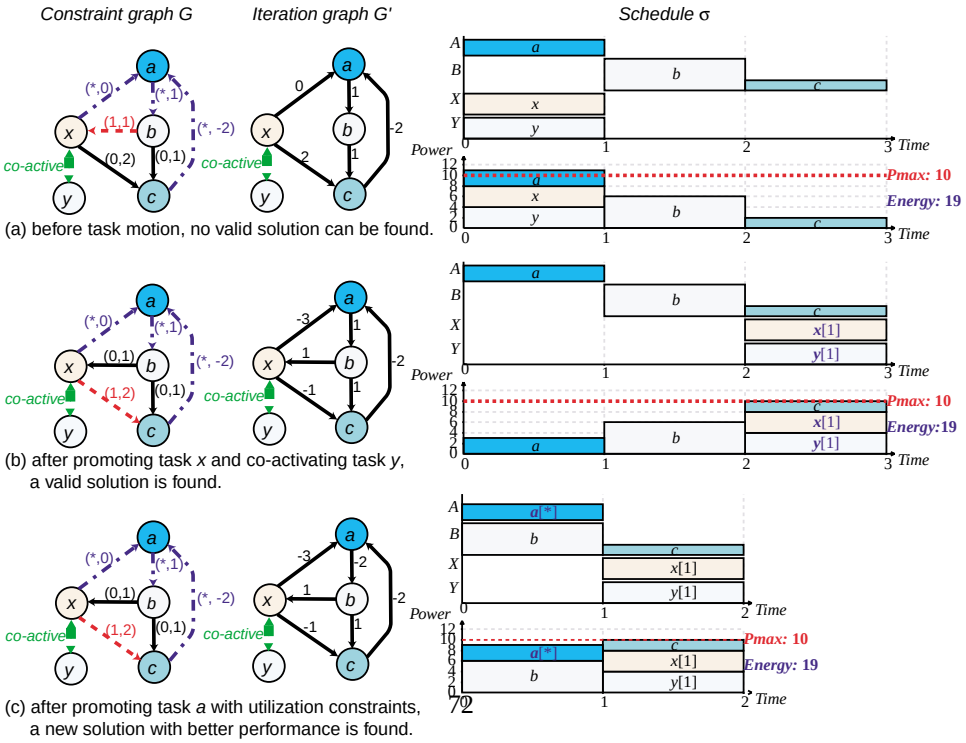


Figure 4.3: Task motion under utilization constraints.

δ is always 0 or 1 that only indicates precedence (data dependency). We capture more general min/max timing constraints that are essential to correctly modeling the operation in new embedded systems, and our approach subsumes DFG as a special case. Second, where existing schedulers use one DFG, we need two graphs: (1) the timing constraint graph G must update dependency depths λ when transforming between intra-iteration and inter-iteration constraints; (2) the iteration graph G' must change the values of corresponding timing constraints δ' in order to correctly reinterpret the new constraints after task motion. Existing techniques do not handle timing constraints, and their δ values never change.

Without loss of generality, we focus our discussion on task *promotion* by which the execution of a task is shifted to the previous iteration of the loop, and the instance of the same task in the next iteration is promoted into the new loop body. The inverse procedure for task *demotion* can be similarly defined.

A task v is *promotable* if either vertex $v \in V$ does not have any incoming forward edges, or all of v's incoming forward edges in G have at least one dependency depth. If σ is a valid schedule of one iteration, we can *promote* a task v according to the *expected loop duration*, which is the finish time τ_σ of σ. Given $\tau = \tau_\sigma$, promoting a task v entails the following transformations on G and G':

1. For each of v's *incoming forward edges* (u,v) in graph G, decrease $\lambda(u,v)$ by one. If (u,v) becomes an intra-iteration constraint, ($\lambda(u,v) = 0$), edge (u,v) is added to graph G' if it is not present in G'.

2. For each v's *outgoing forward edge* (v,u) in graph G, increase $\lambda(v,u)$ by one.

3. For each v's *incoming backward edge* (u,v) in graph G', increase $\delta'(u,v)$ by τ, that is, $\delta'(u,v) = \delta'(u,v) + \tau$.

4. For each v's *outgoing edge* (v,u) in graph G', decrease $\delta'(v,u)$ by τ, that is, $\delta'(v,u) = \delta'(v,u) - \tau$.

Steps 1 and 2 push one dependency depth from v's incoming forward edges to its outgoing forward edges. Step 1 also adds any new intra-iteration constraints after promotion to graph G', which tracks only intra-iteration constraints. Step 3 transforms the incoming backward edges of v for the promotion (its incoming forward edges are managed in step 1). Step 4 transforms the outgoing edges of v, for both forward and backward edges. Steps 3 and 4 can be validated as follows.

When a task v is promoted in graph G', vertex v represents the execution of task v in the next iteration. Therefore, the new start time assignment $\sigma'(v) = \sigma(v) + \tau$. In step 3, before promoting v, edge (u,v) indicates $\sigma(v) - \sigma(u) \geq \delta'(u,v)$. Thus after the promotion, $\sigma'(v) - \sigma(u) = (\sigma(v) + \tau) - \sigma(u) \geq \delta'(u,v) + \tau$. Therefore, the new constraint in G' is $\delta'(u,v) + \tau$. Similarly in step 4, edge (v,u) means $\sigma(u) - \sigma(v) \geq \delta'(v,u)$ before promotion. Thus, $\sigma(u) - \sigma'(v) = \sigma(u) - (\sigma(v) + \tau) \geq \delta'(u,v) - \tau$. The constraint becomes $\delta'(u,v) - \tau$ after the promotion.

When a task v is being promoted, its corresponding min timing constraints (zero or positive values) will become max timing constraints (negative values) by step 3; and vice versa, its corresponding max timing constraints will transform into new min timing constraints by step 4. Promotion effectively reduces the values of min constraints and makes the problem easier to solve by exposing more scheduling opportunities. We say that the constraint is *relaxed*, and this is a key technique for increasing the system's dynamic range.

Fig. 4.2 illustrates task promotion on the example previously shown in Fig. 4.1. Fig. 4.2(a) shows the initial constraint graph G consisting of five vertices representing five tasks a,b,c,x,y. They all have the same execution delay of one time unit, and their power consumption is $p(a) = 3W, p(b) = 6W, p(c) = 2W, p(x) = p(y) = 4W$. Therefore the most power consuming task is b and the least power consuming one is c. Tasks a,x,y have dedicated execution resource A,X,Y ($r(a) = A, r(x) = X, r(y) = Y$), respectively; while tasks b and c share the execution resource B ($r(b) = r(c) = B$). For

brevity, these task attributes are not shown in the graph. The edges in the constraint graph G represent timing constraints. They are denoted as (λ, δ) corresponding to the dependency depths and the values of the timing constraints.

For example, the forward edge (a,b) represents an intra-iteration constraint with dependency depth $\lambda(a,b) = 0$, and it is a min constraint with $\delta(a,b) = 1$ indicating $\sigma(b) - \sigma(a) \geq 1$. Since task a's delay $d(a) = 1$, this constraint can be paraphrased as "task b cannot start until task a completes," that is, tasks a and b must be serialized. Similarly tasks b and c are also serialized by edge (b,c). Edge (x,a) with $\delta(x,a) = 0$ indicates that task a cannot start before task x starts, because $\sigma(a) - \sigma(x) \geq 0$. Edge (x,c) with $\delta(x,c) = 2$ specifies a min separation between task x and task c, that is, $\sigma(c) - \sigma(x) \geq 2$. Therefore, task c must wait until task x has started for two time units. Edge (c,a) with $\delta(c,a) = -2$ is a backward edge representing a max constraint: $\sigma(c) - \sigma(a) \leq 2$. It defines the deadline to start task c relative to the start time of task a. This deadline is equal to the start time of task a plus two time units. In addition to these intra-iteration timing constraints, there is an inter-iteration timing constraint (b,x), indicating that the start time of task b precedes task x in the *next iteration* ($\lambda(b,x) = 1$) by one time unit ($\delta(b,x) = 1$). Inter-iteration constraints are marked as dashed arrows. There is a co-activation dependency between task x and task y. This is denoted as a pair of special timing constraints. As mentioned previously, we assume each iteration must finish within three time units.

The initial iteration graph G' has the same set of vertices representing tasks a,b,c,x,y. The edges in G' only represent intra-iteration constraints. Therefore only constraint value δ' is shown on each edge. Dependency depth λ is not shown since it is always zero in graph G'. For example, the inter-iteration edge (b,x) does not appear in the initial G'. The co-activation dependency is still denoted as a special constraint in G'.

The initial schedule σ computed from the iteration graph G' is also shown in Fig. 4.2(a). It is the same as Fig. 4.1(a). Although all timing constraints are satis-

fied, the schedule σ is not valid since during time [0, 1] the power consumption of the whole system is 11W, exceeding the max power constraint $P_{max} = 10W$. No valid solution is possible even if we try voltage scaling for tasks *x*.

In Fig. 4.2(b) task *x* and its co-activated task *y* are promoted to produce a new valid schedule (same as Fig. 4.1(c), except that the prolog is not shown), which otherwise cannot be achieved without promotion. The constraint graph G will only update the values of dependency depth λ of the timing constraints corresponding to *x*. Since the original schedule finishes at time 3, the timing constraints δ' in G' will be transformed using $\tau = 3$. By step 1, edge $(b,x) \in G$ becomes an intra-iteration edge (solid arrow) and is inserted to G'. By step 2, edges (x,a) and $(x,c) \in G$ become inter-iteration edges (dashed arrows). By step 4, edges (x,a) and $(x,c) \in G'$ reduce their constraint values by $\tau = 3$. Accordingly, task *x*'s outgoing min constraints are transformed into more relaxed max constraints ($\delta'(x,a) = -3, \delta'(x,c) = -1$, compared to 0 and 2 in Fig 4.2(a)). As a result, tasks *x* can be rescheduled in time slot $[2,3]$ without violating any timing constraints, and the max power constraint is also satisfied. Tasks *x* and *y* are promoted together due to co-activation, but they are scheduled as separate tasks because they may not start and finish at the same time.

Fig. 4.2(c) further promotes task *a*. Both graphs G and G' are transformed according to steps 1 – 4. It yields another valid schedule that is a variation of the solution in Fig. 4.2(b). If tasks *b* and *c* are promoted subsequently, the initial constraint graph G in Fig. 4.2(a) will be restored.

4.4.3 Utilization constraints

Task motion is based on the classification of intra-iteration and inter-iteration timing constraints. However, in some cases, it is difficult or unnecessary to decide whether a timing constraint should be intra-iteration or inter-iteration. Such cases are present in the Mars rover. For example, for timing constraints between a heater and a motor

by which the motor is heated periodically, whether to model these constraints as intra-iteration or inter-iteration is not clear. In fact, whether the heaters and the motors stay in the same iteration does not matter. In the computation domain, these correspond to background, preemptible tasks that need not synchronize with the main control loop but must be given a share of the CPU time to avoid starvation.

We call such constraints *utilization*-based timing constraints. They can be expressed as either intra-iteration or inter-iteration ones. A utilization constraint between two tasks u and v is also represented as an edge $(u,v) \in E$ in constraint graph G with its dependency depth denoted as $\lambda(u,v) = *$, indicating that it can be either zero or non-zero.

Now we examine task motion under utilization constraints. It needs only minor modifications to the procedure we defined in Section 4.4.2.

(a) The initial iteration graph G' will include both intra-iteration constraints and *utilization constraints* in its edges. *(Treat utilization constraints as intra-iteration).*

(b) A task v is promotable if either vertex $v \in V$ does not have any incoming forward edges, or the dependency depths λ of all v's incoming forward edges are positive values *or* $*$. *(Treat utilization constraints as inter-iteration).*

(c) The modified procedure for promoting a task v is as follows.

1. For each of v's incoming forward edges (u,v) in graph G, decrease $\lambda(u,v)$ by one, *if* $\lambda(u,v) \neq *$. If $\lambda(u,v)$ becomes 0, add edge (u,v) to graph G' if it is not present in G'. *(No update for utilization constraints in step 1).*
2. For each v's outgoing forward edge (v,u) in graph G, increase $\lambda(v,u)$ by one, *if* $\lambda(u,v) \neq *$. *(No update for utilization constraints in step 2).*
3. For each v's incoming backward edge (u,v) in graph G', $\delta'(u,v) = \delta'(u,v) + \tau$, *if* $\lambda(u,v) \neq *$. *Otherwise,* $\delta'(u,v)$ *remains unchanged. (Do nothing for utilization constraints in step 3).*

4. For each v's outgoing edge (v,u) in graph G', $\delta'(v,u) = \delta'(v,u) - \tau$. *(Same as previous step 4)*.

Since utilization constraints can be either intra-iteration or inter-iteration, by giving them some special treatments, the modified procedure is straightforward except steps 3 and 4 need more explanation. In step 3, if edge (u,v) represents a utilization constraint, $\delta'(u,v)$ can be transformed into either one of the two forms: $\delta'(u,v)$ or $\delta'(u,v)+\tau$, since it can be either intra-iteration or inter-iteration. That is, the transformation is valid either $\sigma'(v) - \sigma(u) \geq \delta'(u,v)$ or $\sigma'(v) - \sigma(u) \geq \delta'(u,v) + \tau$ holds. Obviously, the solution to these two inequalities with an *OR* relation is $\sigma'(v) - \sigma(u) \geq \delta'(u,v)$, which means the constraint with the smaller value applies. The value of a utilization constraint will not increase by τ. Likewise, in step 4, the value of the new constraint is the smaller one between $\delta'(v,u) - \tau$ and $\delta'(v,u)$, which is $\delta'(v,u) - \tau$. In summary, if the promoted task v has any incoming utilization-constraint edges, these edges remain the same in the iteration graph G' during the promotion. For v's outgoing utilization-constraint edges, the values of constraints in G' are decreased by the loop duration τ. As a result, utilization constraints will always be relaxed to produce more scheduling opportunities.

For example, if resource A is a heater, a motor, or a CPU running a preemptible background tasks, then we can model task a with utilization constraints. The modified procedure for task motion under utilization constraints in illustrated in Fig. 4.3.

Fig. 4.3(a) shows the initial graphs G, G' and schedule σ. It is identical to Fig. 4.2(a), except that utilization constraints are marked as a different type of dashed arrows in the constraint graph G, and their dependency depth $\lambda = *$. Since the iteration graph G' is the same as the graph G' in Fig. 4.2(a), no valid schedule could be found. To address this problem, Fig. 4.3(b) shows promotion to tasks x and y. It is similar to Fig. 4.2(b) except the utilization constraint (x,a) is not updated in graph G. It shows a valid schedule, which is the same as schedule in Fig. 4.2(b).

In Fig. 4.3(c), when task a with utilization constraints is promoted, the corresponding constraint values in graph G' are different from those in Fig. 4.2(c) in comparison. Specifically, by modified step 3, utilization constraint (c,a) will not increase its value in G'. $\delta'(c,a)$ will remain -2 as opposed to 1 in Fig. 4.2(c). The same rule also applies to utilization constraint (x,a) such that $\delta'(x,a) = -3$ instead of 0. Since the serialization chain formed by min constraints is broken, tasks a,b,c (after promoting a, the chain becomes b,c,a in Fig. 4.2(c)) no longer need to be serialized. Now task a, a small power consumer, can overlap with b such that an unexpected solution with a shorter execution time ($\tau = 2$) is discovered, and it also satisfies the max power constraint. This optimal solution could not have been obtained without using utilization constraints, which enable more aggressive, provably correct relaxation of the time constraints.

4.5 Scheduling Algorithms

Given a scheduling problem, the scheduler to support power management decisions must compute a schedule σ that meets goals in multiple dimensions. First, σ must be time-valid in that all timing constraints, including intra-iteration, inter-iteration and utilization-based, are satisfied. Second, it must be power-valid for a max power constraint with a reduced energy cost for a min power constraint. Finally, the scheduler must evaluate different versions of the loop iterations to either improve the schedule with shorter execution time or less energy cost, or explore various power/performance trade-offs.

We present our power-aware task motion technique as follows. In Section 4.5.1 we build an iteration graph G' that tracks only the intra-iteration constraints from the constraint graph G. The promotion to one task transforms both graphs G and G', to be presented in Section 4.5.2. Section 4.5.3 introduces the power-aware scheduler with task motion technique that evaluates different versions of the loop by using a power-aware scheduler to compute a single-iteration schedule for each version. We derive the

```
ITERATION GRAPH(graph G)
    create graph G'(V,E), with G'.V := G.V, G'.E := ∅
    for each edge (u,v) ∈ G.E loop
        if (λ(u,v) = 0 or λ(u,v) = *) then
            add edge (u,v) to G'.E, with δ'(u,v) := δ(u,v)
        end if
    end loop
    return G'
```

Figure 4.4: Algorithm to construct the iteration graph.

scheduling algorithms presented in [44] as the power-aware scheduler to reduce energy cost. After the best version is selected with the minimum energy cost, the scheduler computes a prolog and an epilog to start and finish the loop execution. Other solutions with different loop durations, as well as those that cannot be evaluated together the existing version, are recorded for further evaluation.

4.5.1 Construction of the iteration graph

The concept of the iteration graph is introduced in Section 4.4. The algorithm in Fig. 4.4 constructs an iteration graph G' based on a constraint graph G with intra-iteration, inter-iteration and utilization-based constraints.

4.5.2 Task promotion algorithm

We present two algorithms for task promotion and the corresponding graph transformation to both constraint graph G and the iteration graph G'. The algorithm in Fig 4.5 decides whether a task v is promotable by checking v's incoming forward edges. If they consist of only inter-iteration and utilization-based constraints, or if v does not have any incoming forward edges, then v is promotable. The algorithm in Fig 4.6 promotes a task v by transforming both graphs G and G' with an expected loop duration τ.

```
PROMOTABLE(graph G, vertex v)
    for each v's incoming forward edge (u,v) ∈ G.E loop
        if (λ(u,v) = 0) then
            return FALSE
        end if
    end loop
    return TRUE
```

Figure 4.5: Algorithm to decide whether a task v is promotable.

```
PROMOTE(graph G, graph G', vertex v, time τ)
    for each v's incoming forward edge (u,v) ∈ G.E loop # step 1
        if (λ(u,v) ≠ ∗) then
            λ(u,v) := λ(u,v) − 1
        end if
        if (λ(u,v) = 0) then
            add edge (u,v) to G'.E with δ'(u,v) := δ(u,v)
        end if
    end loop
    for each v's outgoing forward edge (v,u) ∈ G.E loop # step 2
        if (λ(v,u) ≠ ∗) then
            λ(v,u) := λ(v,u) + 1
        end if
    end loop
    for each v's incoming edge (u,v) ∈ G'.E loop # step 3
        if (λ(u,v) ≠ ∗ and δ(u,v) < 0) then
            δ'(u,v) := δ'(u,v) + τ
        end if
    end loop
    for each v's outgoing edge (v,u) ∈ G'.E loop # step 4
        δ'(v,u) := δ'(v,u) − τ
    end loop
    return
```

Figure 4.6: Task promotion algorithm.

4.5.3 Algorithm for power-aware task motion/scheduling

The algorithm is shown in Fig. 4.7. It first constructs a iteration graph G' from the constraint graph G. Then G' is scheduled by a power-aware scheduler, which is derived from [44]. The returned schedule σ is kept as a temporarily best schedule and whose duration τ_σ is taken as the expected loop duration τ. Then the algorithm traverses all vertices in $G.V$ in a topological order by extracting one promotable task v at each step. When a task v is promoted, both graphs G and G' are updated. Then the power-aware schedule is invoked again to examine whether an improved schedule with the same execution time and less energy cost can be found, and the better schedule is stored. In case a schedule with a different finish time is found, it indicates that another version of the loop. It is not appropriate to simply discard the slower schedule, because it could represent a different power/performance trade-off. Instead, the graphs leading to the incompatible versions are stored in set Alt, and the algorithm cancels the last promotion and attempts another topological ordering. The algorithm completes if all tasks are promoted, or the topological traversal cannot proceed since the next promotion always generates an incompatible version. Finally, it computes two additional schedules, one for the prolog and one for the epilog (algorithms not shown). Once the algorithm finds the best schedule σ for the loop body, it returns the full set of schedules that includes σ_{prolog}, σ, and σ_{epilog}. The algorithm also returns the set Alt that contains graphs leading to alternative solutions. These graphs will be examined by the same algorithm to evaluate different power or energy vs. performance trade-offs.

4.6 Experimental Results

We use the NASA/JPL Mars rover [4] to evaluate the effectiveness our power-aware task motion technique. We first construct a system-level representation that includes the mechanical and thermal subsystems, as well as different energy sources. Then, we

```
POWER AWARE TASK PROMOTION(graph G, P_max, P_min)
    G0 := G; Alt := ∅
    G' := ITERATION GRAPH(G)
    σ := POWER AWARE SCHEDULING(G', P_max, P_min) # [44]
    Ec := Ec_σ(P_min); τ := τ_σ
    V' := G.V; V_prolog := ∅
    for each v ∈ V' loop
        if PROMOTABLE(G, v) then
            V' = V' − {v}
M:          PROMOTE(G, G', v, τ)
            if (G ∈ Alt) then
                break
            end if
            σ' := POWER AWARE SCHEDULING(G', P_max, P_min)
            if (τ_σ' ≠ τ) then
                Alt := Alt + {G}; V' = V' + {v}
                undo step M
            else
                if (Ec_σ' < Ec) then
                    σ := σ'; V_prolog := V'
                end if
            end if
        end if
    end loop
    if (V_prolog = G.V or V_prolog = ∅) then
        return σ, ∅, ∅, Alt
    end if
    V_epilog := G.V - V_prolog
    σ_prolog = PROLOG(G0, V_prolog, σ)
    σ_epilog = EPILOG(G0, V_epilog, σ)
    return σ, σ_prolog, σ_epilog, Alt
```

Figure 4.7: Power-aware task motion algorithm.

examine the results after applying our scheduling techniques.

4.6.1 A system-level constraint model of the Mars rover

The rover travels between different target locations on the Mars surface to perform scientific experiments and shoot images. Its power sources consist of a non-rechargeable battery and a solar panel. The life-time of its mission is limited by the amount of remaining battery energy. Since the temperature on Mars surface can be as low as -80°C, the rover must heat its motors periodically as it drives them to move. Thus, mechanical and thermal subsystems are the major power consumers.

Our model captures timing constraints across different resources including computational, thermal and mechanical subsystems. We focus on a typical operating condition when the rover is traveling. When the rover drives its six wheels for a full rotation, it is called one step, which is about 7cm in distance. Before driving the wheels, it must first detect any obstacles on its way and choose a safe angle to turn. Then it turns itself in the correct direction using the four steering motors. Finally, the six wheel motors are driven. Therefore, hazard detection, steering, and driving must operate in sequence. Other constraints are related to heating the motors in a certain period prior to driving them, as summarized in Table 4.1. We assume the power consumption of tasks varies with environmental temperature that tracks the sunlight intensity, and we investigate three scenarios with different solar power output: 14.9W (noon time), 12W, and 9W (dusk). The max power constraint is equal to the available solar power plus 10W maximum battery power output. We also extract the solar power level as the min power constraint to distinguish the free power from the costly power. Table 4.2 illustrates the power sources and consumers in three scenarios.

The constraint graph for the Mars rover is shown in Fig. 4.8. During each iteration, the rover moves two steps (14cm). We assume all heaters are independent resources and one heater can heat two motors at a time. Therefore there are a total of five thermal

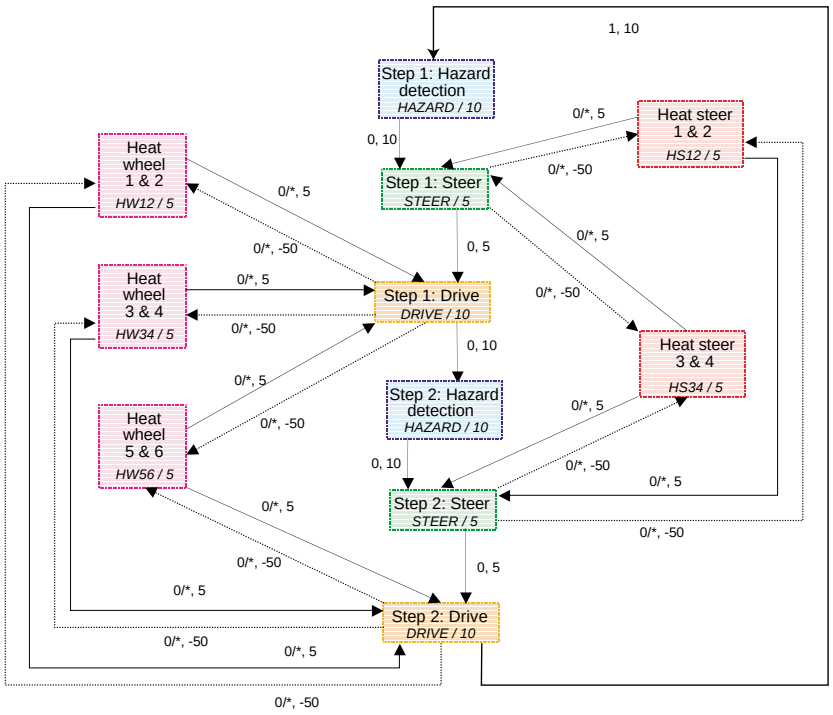


Figure 4.8: Constraint graph of the Mars rover.

Operation	Duration(s)	Timing constraints
Heating steering motors	5	At least 5s, at most 50s before steering
Heating wheel motors	5	At least 5s, at most 50s before driving
Hazard detection	10	At least 10s before steering
Steering	5	At least 5s before driving
Driving	10	At least 10s before next hazard detection

Table 4.1: Timing constraints of the Mars rover.

Power sources & tasks	Duration (s)	Power (W)		
		Best case @-40 °C	Typical case @-60 °C	Worst case @-80 °C
Solar panel		14.9	12	9
Battery pack		10 max	10 max	10 max
CPU	constant	2.5	3.1	3.7
Heating two motors	5	7.6	9.5	11.3
Driving	10	7.5	10.9	13.8
Steering	5	4.3	6.2	8.1
Hazard detection	10	5.1	6.1	7.3

Table 4.2: Power sources and consumers of the Mars rover.

heaters. Four steering motors are considered a single steering mechanical resource. The six wheel motors are modeled as one mechanical unit for driving. There is also a laser guided digital component for hazard detection. Each task v is denoted with its resource mapping $r(v)$ and its execution delay $d(v)$. The power consumption is not shown since it varies in different scenarios. Each edge (u,v) is denoted with its dependency depth $\lambda(u,v)$ and timing constraint $\delta(u,v)$. The timing constraints on the heating tasks are actually utilization-based constraints. They are denoted differently from inter-iteration constraints and intra-iteration ones. We first treat them as intra-iteration and then change them to utilization constraints to compare the differences in their results.

4.6.2 Scheduling results

We use the energy cost to non-rechargeable battery $Ec_\sigma(P_{min})$ and the execution time (τ_σ) as metrics to examine the scheduling results by the following techniques:

(0) the existing manual solution,

(I) previous power-aware scheduling [44],

(II) power-aware task motion without utilization constraints,

(III) power-aware task motion with utilization constraints.

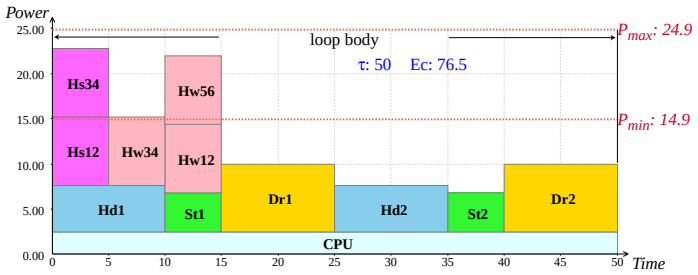
We first evaluate the scheduling results in three individual scenarios with different power constraints. Then, we present a case study by combining the three scenarios into one comprehensive scenario, where the power constraints vary over time.

Scenario 1: high power budget, P_{max} = 24.9W, P_{min} = 14.9W

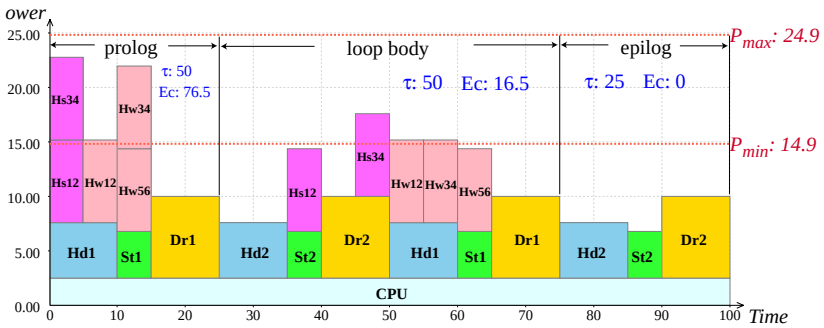
Fig. 4.9(a) shows the power-aware schedule (I) for this scenario. In this scenario, since the power budget is sufficient, some tasks are executed in parallel, and thus the schedule is fast. However, some energy cost (76.5J) is drawn at the beginning of the schedule while the solar energy is under-utilized in the latter part. Without task motion, we cannot further exploit free solar power to reduce energy cost.

Fig. 4.9(b) shows the schedule after power-aware task motion(II), though without exploiting utilization-based constraints. Some heating tasks are promoted such that they consume free solar energy instead of costly battery energy. The resulting performance is the same as the previous schedule (50s), but the energy cost is significantly reduced (16.5J). In this schedule, the timing constraints on heaters are considered as intra-iteration constraints.

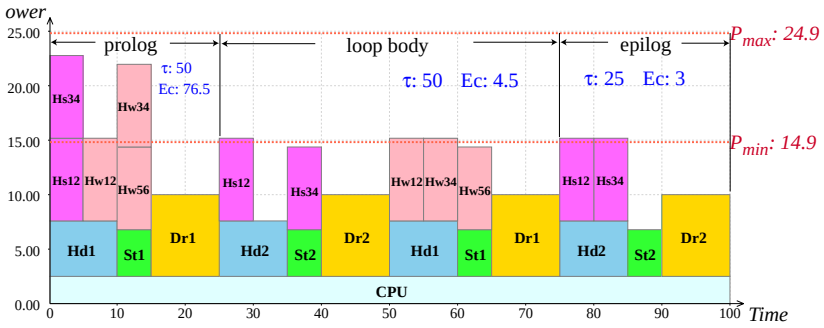
If we consider utilization-based constraints (III), we can further improve the schedule significantly, as shown in Fig. 4.9(c). The heating tasks are reordered to other slots



(a) Power-aware schedule (I)



(b) Power-aware task promotion without utilization constraints (II)



(c) Power-aware task promotion with utilization constraints (III)

Figure 4.9: Schedule for Scenario 1 (highest power budget).

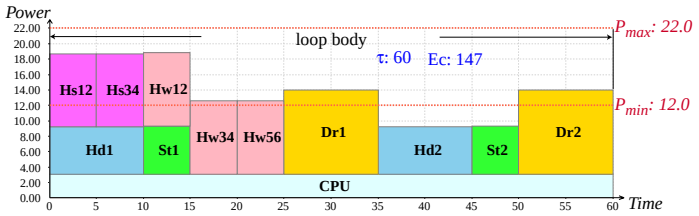
with even less energy cost to non-rechargeable battery. As a result, the schedule in Fig. 4.9(c) is strictly better than the previous two schedules in the sense that it delivers same performance (50s) with less energy cost (4.5J). This superior schedule could not have been found until we converted the constraints on the heating tasks to utilization edges. In general, utilization constraints can expose rich new scheduling opportunities by effectively increasing the number of alternative time intervals for partially reordering tasks. They are useful for a power manager to either minimize energy cost or evaluate different energy/performance trade-offs.

Scenario 2: moderate power budget, P_{max} = 22W, P_{min} = 12W

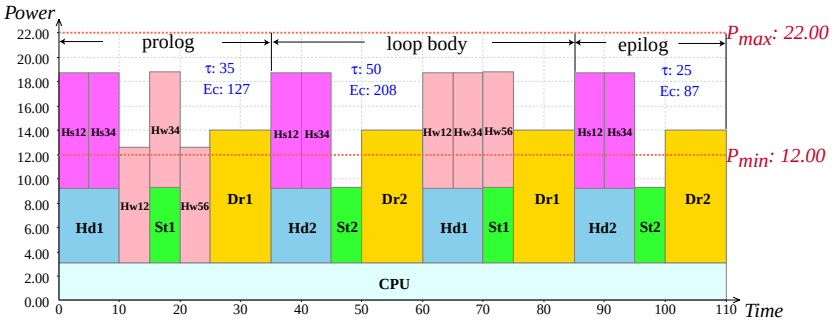
Fig. 4.10(a) shows the power-aware schedule (I) for this scenario. With a smaller power budget and a reduced level of free (solar) power source than Scenario 1, this new schedule is slower (τ = 60s) while drawing more energy from the battery (147J). It is notable that task motion does not yield a different schedule if we model the constraints on heating tasks as intra-iteration ones. A somewhat surprising result (III) can be discovered if the scheduler exploits the utilization constraints, as shown in Fig. 4.10(b). The resulting schedule can be as fast as the schedule found in Scenario 1 (τ = 50s), if paying a higher energy cost (208J) is acceptable. Neither solution is strictly better than the other, since they represent alternative design points for energy/performance trade-offs. Again, conversion to utilization constraints exposes more aggressive but safe design points that otherwise would not be possible.

Scenario 3: low power budget, P_{max} = 19W, P_{min} =9W

Fig. 4.11 shows a slow schedule (0) for this scenario. A tight power budget forces all operations to be serialized, leading to a low-performance (τ = 75s) and high-cost (Ec = 388J) schedule. Since overlapping any two tasks will violate the max power budget, task motion cannot yield any alternative schedule.



(a) Power-aware schedule (I)



(b) Power-aware task motion with utilization constraints (III)

Figure 4.10: Schedule for Scenario 2 (moderate power budget).

Table 4.3 summarizes the scheduling techniques that are applied to the three scenarios. It shows that our power-aware task motion technique and utilization constraints can support more aggressive design space exploration effectively.

A comprehensive scenario: the available power varies over time

The existing schedule used in the past mission followed a low-power design paradigm. To avoid exceeding max power budget, the designers at JPL implemented a fully serial-

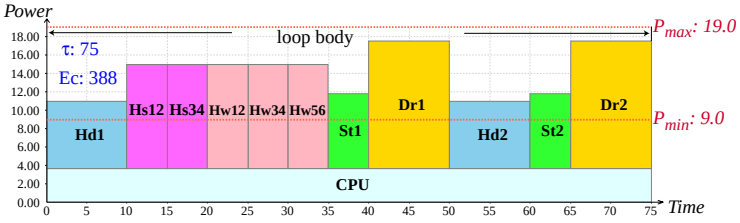


Figure 4.11: The serial schedule for Scenario 3 (lowest power budget).

Scenario	(0) JPL's Low-power (hand-craft)	(I) Power-aware	(II) Power-aware + Task motion	(III) Power-aware + Task motion + Utilization constraint
1	τ = 75s Ec = 0J √	τ = 50s Ec = 79.5J ×	τ = 50s Ec = 16.5J ×	τ = 50s Ec = 4.5J √
2	τ = 75s Ec = 55J √	τ = 60s Ec = 147J √	same as (I)	τ = 50s Ec = 208J √
3	τ = 75s Ec = 388J √	same as (0)	same as (0)	same as (0)

√ = keep × = drop

Table 4.3: Comparison of schedules in a three scenarios.

Time frame (s)	Scenario	JPL (0-0-0)			Task motion A (III-I-0)			Task Motion B (III-III-0)		
		Distance (step)	Time (s)	Energy cost (J)	Distance (step)	Time (s)	Energy cost (J)	Distance (step)	Time (s)	Energy cost (J)
0 - 599	1	16	600	0	24	600	129	24	600	129
600 - 1199	2	16	600	440	20	600	1470	23	600	2482
1200 -	3	16	600	3114	4	150	776	1	10	85
Total		***48***	***1800***	***3554***	***48***	***1350***	***2375***	***48***	***1210***	***2696***
Improve-ment						***33%***	***33%***		***49%***	***24%***

Table 4.4: Comparison of schedules in a comprehensive scenario.

ized schedule (0) that was fixed in all conditions, without tracking the available power budget including the solar source. It is identical to our schedule in Scenario 3 with the lowest power budget. Without our task motion technique that aggressively explores the design space, the designers had no alternative choices for different scenarios but over-constrained the existing design for the worst case. However, the existing low-power solution draws less costly energy from the battery than our solutions. Our schedules, on the other hand, speed up the rover's movement by up to 50% in Scenario 1 with the maximum power budget (III). In Scenario 2, we have two alternative solutions that improve the rover's performance by 25% (I) and 50% (III), respectively. However, our faster schedules draw more costly energy from the battery. To evaluate this trade-off between performance and energy cost, we apply our schedules to a scenario where the available solar power varies over time.

Suppose the mission is to travel to a target location in a distance of 48 steps. The

mission starts with maximum solar power at 14.9W (Scenario 1). Then, it drops to 12W (Scenario 2) after 10 minutes, then falls to 9W (Scenario 3) 10 minutes later. If the existing serial schedule (0) (Fig. 4.11) is applied, the rover will spend 10 minutes evenly in the three scenarios, since it has a fixed moving speed (16 steps per 10 minutes). This results in a long execution time (30 minutes) and considerable energy cost in Scenario 3. On the other hand, our power-aware scheduler can produce two schemes. In scheme A, the rover finishes 50% of its work (24 steps) in the first 10 minutes by using our schedule (III) for Scenario 1 (Fig. 4.9(c)). Then it completes 42% of the work (20 steps) in the next 10 minutes by schedule (I) for Scenario 2 (Fig. 4.10(a)), leaving the remaining 8% (4 steps) in Scenario 3 for only 2.5 minutes. In scheme B, the rover also finishes 24 steps in the first 10 minutes with the same schedule (III) for Scenario 1. By using the fast schedule (III) (Fig. 4.10(b)) for Scenario 2, the rover almost completes the whole mission by traveling 23 steps in the next 10 minutes, leaving the last step in Scenario 3 for only 10 seconds (because its prolog is long). Since our schedules accelerate the execution with sufficient power budget in first two scenarios, the rover can finish the mission earlier before having to work in the costly Scenario 3. The analysis of this case study in Table 4.4 shows that both power-aware schemes A and B are strictly better than the existing design in that they win performance and energy saving simultaneously. Scheme A delivers a higher performance with a 33% improvement, while saving 33% of the costly energy from non-rechargeable battery. Scheme B even further speeds up the execution by 49% with a 24% energy reduction compared with the existing solution. Moreover, these two alternative designs with different power/performance trade-offs are discovered by our automated scheduling techniques. They cannot be extracted otherwise by the existing techniques.

4.7 Chapter Summary

We have presented a power-aware task motion technique for enhancing the dynamic range of embedded systems powered by heterogeneous energy sources that include renewable, unsteady ones like solar panels. They must be able to not only operate as low-power devices when the supply power is low, but equally importantly use the free abundant energy for useful work while respecting power and timing constraints. We used a DVS Anomaly example to show the pitfalls of applying existing power management techniques without considering system-level dependencies like co-activation, and this has resulted in not only higher energy consumption but also violation of max power constraints. We then showed our constraint formulation and task motion to safely transform the tasks while respecting these system-level dependencies. We further enhanced task motion by exploiting utilization-based constraints that exposed additional scheduling opportunities for preemptible, background tasks or even non-computational power consumers such as heaters. These all served to enhance the dynamic range while ensuring all transformations are safe and provably correct. Experimental results on the Mars rover demonstrated the effectiveness of our approach for the solar- and battery-powered system. We expect the benefits to transfer to a whole emerging class of new embedded systems that must draw energy from many renewable but unsteady sources.

Part III

Data Regular Scheduling

Chapter 5

SuperDVS

Dynamic voltage scaling (DVS) is a popular approach to power and energy reduction in microprocessors: it not only reduces the average power level, but also increases the processor's energy efficiency (i.e., lower energy per instruction). However, DVS will not be a fruitful technique for further energy reduction based on today's single processor assumption. In fact, it may actually lead to higher energy consumption due to the lack of system-level considerations. We propose to exploit system-level parallelism as an effective means to achieving the next order of energy improvement for the entire system. By using multiple processors to perform the same task, each processor has reduced workload and thus can run at even more energy efficient points beyond the limits of today's best DVS without sacrificing performance. A secondary effect is that the smoother power profile and reduced peak current not only reduce time/power overhead associated with voltage scaling, but also extend battery life. Experimental results on an image processing algorithm show a 60% reduction in energy and 80% in power compared to the best DVS approach.

5.1 Introduction

Dynamic voltage scaling (DVS) is a well-studied technique for minimizing energy consumption in embedded microprocessors. Many modern processors are designed to operate at different voltage and frequency settings as an effective way to manage power usage. It is advantageous to operate the processor at a lower voltage whenever possible, because energy is proportional to V^2. In other words, it takes less energy to execute an instruction at a lower voltage, even though the total execution time is longer. DVS techniques exploit slacks in the task set by slowing down the processor just enough without violating the deadline. Many DVS techniques have been proposed to overcome the limits in slowing down the processor, including inter-task DVS, intra-task DVS, slack borrowing, etc.

5.1.1 Limits of DVS

DVS has been studied extensively for single processors. However, DVS is running against several fundamental limits. Due to variation in workload, it is not always possible to run the processor at the lowest possible constant speed at all times; for example, different types of MPEG frames require a wide range of processing. This not only prevents the processor from operating at the optimal rate, but also forces the processor to pay power and timing overhead associated with voltage scaling. In fact, as processor performance increases, the voltage scaling granularity decreases, and as a result, as much as 30% of the energy could be spent on scaling overhead alone. Unfortunately, most DVS techniques proposed to date do not consider any overhead.

Another limit is DVS's assumption about a single processor, even though many realistic system include peripheral devices controlled by the processor, but this dependency is never modeled. In fact, the power consumption at the system level may be dominated by these peripherals, rather than the processor, and it is crucial to exploit shutdown opportunities (especially communication interface) in peripherals. However,

most peripherals are not as power manageable as the processor. As a result, the DVS approach to increasing the processor's power efficiency at the expense of performance can actually keep the dependent peripherals in the same high-power operating mode for a longer period of time. That is, not only is the power saving due to DVS limited to its percentage contribution to the entire system, it can actually result in higher overall energy consumption.

5.1.2 Beyond DVS limit

To further improve energy efficiency beyond DVS, we break the limits of DVS described above. First, we must improve power efficiency without sacrificing performance; otherwise, we will repeat the pitfall of increasing energy consumed by dependent peripherals. Second, we must decrease the dynamic range of the power of the processor, even though the workload has a wide dynamic range, in order to maximize its energy efficiency (in terms of Joules per instruction). This can be achieved by exploring the *granularity* for voltage scaling, where coarser grain scaling results in lower overhead.

In this chapter, we propose to break the DVS limits by exploring system-level pipelining. We improve energy efficiency by applying voltage scaling to each code partition, but make up for the performance loss with parallelism. We make the observation that n processors running at $1/n$ speed will be significantly more energy efficient than a single processor running at the speed determined by today's best DVS techniques. It not only significantly reduces the energy consumed by the processor, but also results in much lower peak power and a smoother power profile, both of which are attractive features for batteries. We also determine the optimal granularity for voltage scaling so as to balance adaptivity with the adaptation cost.

The grand challenge will be to extract parallelism in the application so that it can be mapped onto a multi-processor architecture with a much higher power efficiency

as a whole system. Parallelism extraction from a sequential program like C is a very difficult problem in general, but fortunately data regular applications can be described in a variety of data flow models that are amenable to mapping onto architectures with multiple processors. Another challenge is to design the architectural features to enable the processors to compose with each other efficiently.

This chapter uses an automatic target recognition (ATR) example to demonstrate the next order of magnitude power/energy saving beyond DVS for data regular applications. We show our parallelized code with shared memory communication, and we discuss the software run-time support in the form of adjusting references for each pipeline stage.

5.2 Related Work

Dynamic voltage scaling technique has been studied extensively recently. Researchers have addressed DVS related issues in the following aspects.

Real-time scheduling has been extended to DVS scheduling on variable-voltage processors. A few analytical models have been proposed. Weiser et al. proposed the initial scheduling model in [71], and the aspect of energy minimization was analyzed by Yao et al. and the optimal off-line schedule is given in [74]. Hong et al. proposed an off-line scheduling heuristic for non-preemptive systems in [25] and an on-line algorithm for mixed workload of both sporadic and periodic tasks in [27]. Ishihara et al. analyzed the optimal schedule for a processor that can operate in several discrete voltages in [34] and proposed an integer linear programming solution. Okuma et al. proposed a DVS scheme [48] that always guarantees deadlines for all tasks, although the energy may not be optimal. Shin et al. presented a run-time checking mechanism that can shut down the processor or adjust the processor speed [59] and an algorithm to minimize energy for periodic tasks [60]. Quan et al. improved this technique by finding an optimal schedule for both periodic and sporadic tasks [54].

More realistic DVS models have been proposed to consider design issues, e.g. DVS overhead and the presence of the scheduler. Hong et al. presented a synthesis design flow for variable-voltage processor cores [28]. The impact of DVS overhead is studied with a few scheduling schemes. It turns out that the analytical models may not be yield optimal solutions when overhead is taken into account. Burd et al. presented an implementation scheme for a microprocessor with DVS capability [13]. In [52] Pering et al. simulated different DVS scheduling algorithms and considered the presence of the scheduler as a part of the system, and the DVS overhead is also examined.

Some researchers have applied DVS to embedded applications, where the processor only deals with one or a few specific tasks. Shin et al. proposed an intra-task DVS scheme [58] that tries to maximally utilize the slack time within one task, as opposed to inter-task DVS scheme that borrows slack time from finished tasks. Im et al. proposed an inter-task DVS scheme for multi-media applications in [32] based on the observation that the slack time cannot be utilized when no task is available. The solution is to buffer the tasks such that the slack time can be used when the buffer is not empty.

5.3 Motivating Example: ATR

We use an automatic target recognition (ATR) algorithm [61] as our motivating example. Its block diagram is shown in Fig. 5.1. The algorithm is described in Fig. 5.2. It takes a sample image as the input. The TARGET DETECTION module detects M targets on the original image. For each target, a region of interest (ROI, which is a smaller image surrounding the target on the original image) $roi0$ is extracted and Fourier transformed by FFT module to space-frequency domain. The transformed $roi1$ is multiplied by a few predefined templates, then Fourier transformed inversely by IFFT module back to space domain. For each template, the resulting $roi2$ has a specific attribute Val that indicates the distance of the target. If it is larger than a threshold value, $roi2$ is fed to the COMPUTE DISTANCE module, an image processing routine to calculate the

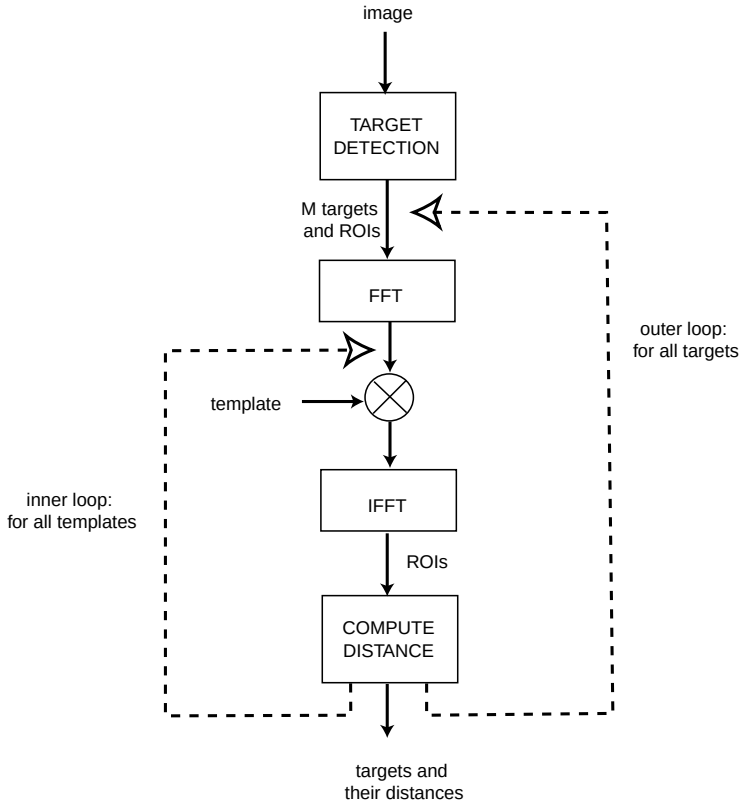


Figure 5.1: Block diagram of the ATR algorithm.

distance.

The timing requirement to the ATR algorithm is to sustain a frame rate, the number of images processed per unit time. Its inverse is called *frame delay*, which is the maximum delay time to process each frame. The goal is to reduce the energy consumption of the processor for a given frame delay.

One key characteristic of the algorithm is that its execution delay falls in a diverse range with regard to M, the number of targets detected by TARGET DETECTION module, since FFT and IFFT are quite computation-intensive. In addition, the *if* statement at label B also affects the distribution of the execution delay due to another computation-intensive module COMPUTE DISTANCE.

Among the current DVS techniques, *intra-task DVS* [58] is most suitable for this problem. By applying this technique, static timing analysis is performed at compile time to insert additional annotations at labels A and B to compute remaining the worst-case workload (in worst-case instruction count, *WCIC*). In the beginning, the remaining *WCIC* is the *WCIC* of the whole algorithm. At A, the remaining *WCIC* is computed based on number of targets M. At B, the outcome of the conditional branch will modify *WCIC* by subtracting the instruction count (*IC*) of COMPUTE DISTANCE module. The *ICs* of other modules are fixed for fixed-size images, e.g., 128×128. Therefore, A and B are the only places where the remaining *WCIC* could decrease.

During the execution of the algorithm at run-time, when a new frame arrives, the processor speed is set based on the frame delay and the *WCIC* of the whole algorithm. As the program execution reaches A or B, the remaining *WCIC* is updated and the processor adjusts its frequency and voltage based on the remaining time and *WCIC*.

It is notable that the voltage at point B could scaled made many times for each image frame, since B is located inside the inner loop. In our current analytical study, we set the maximum number of targets to 7, so there might be 0-7 targets in each frame. The number of matching templates is 3. As a result, B could be reached up to

```
ATR(image im, array TEMPLATE, float Val_th)
    TARGET := TARGET DETECTION(im)
A:  for each (target ∈ TARGET) loop
        roi0 := getRoi(im, target)
        roi1 := FFT(roi0)
        for each (template ∈ TEMPLATE) loop
            roi2 = IFFT(roi1 × template)
B:          if roi2.Val > Val_th then
                COMPUTE DISTANCE(roi2)
            end if
        end loop
    end loop
    format data and return computed distance
```

Figure 5.2: The ATR algorithm.

21 times for one frame. Therefore, to process each frame, the processor may change its voltage/frequency up to 23 times, including the initial maximum speed setting and the speed drop after A.

A typical power profile for intra-task DVS is shown in Fig. 5.3. It always starts with a high power spike at the beginning of each time slot for one frame, and then steps down once at A, followed by a few more drops each time when the *if* condition evaluates to *FALSE* at B.

Intra-task DVS appears to be the right solution for reducing processor energy for application-specific algorithms. However, there are still some issues that need to be addressed for more energy efficient designs. For example, there is a high power spike at the beginning of each time slot, because the initial processor speed is always set at a high speed for the worst case. In most cases the worst-case instruction count *WCIC* is not a good measure of the typical workload of the application. Therefore, what is seen in the power profile chart is a few sharp slumps after the initial spike at peak power. By the time when one frame is about to be finished, the processor normally operates at a very low power level; and it must switch to high power for the next frame in anticipation for the worst-case workload.

A power profile in such a pattern has several undesirable properties. High peak

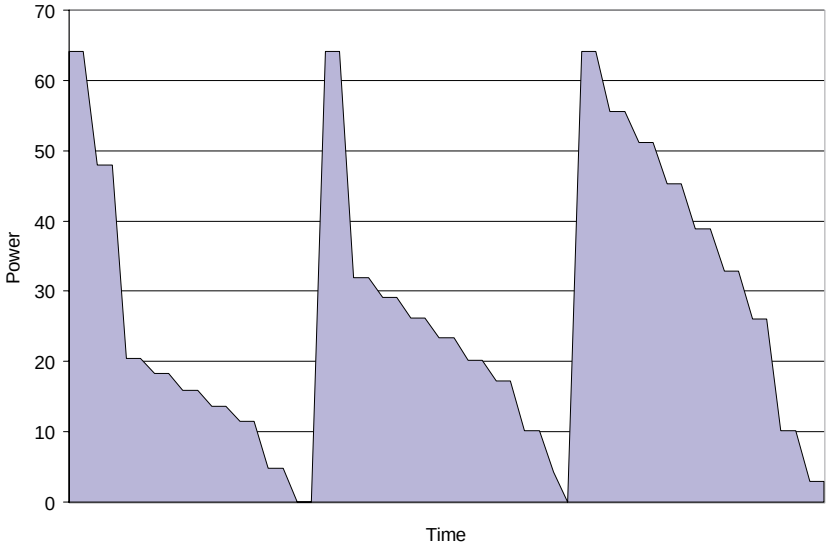


Figure 5.3: Power profile of intra-task DVS.

power is known to have many negative effects, e.g. it shortens the battery life. In addition, frequent switching between high-power mode and low-power mode will incur an overhead on both time and energy when the processor is changing its supply voltage and frequency. Although most DVS studies do not consider this overhead, it can be non-trivial when the voltage change is significant. Finally, the power profile with high jitters is not very energy efficient compared to a smooth one.

One potential solution to this problem is to extract the parallelism in the algorithm and distribute the workload into multiple processors. Successful parallelization can reduce both power and energy by running only a part of the code on each processor at a slower speed. Even though more processors are involved, the overall energy and power consumption will still be reduced significantly. However, it is generally difficult to parallelize serial algorithms. In addition, parallel algorithms require sophisticated support for synchronization, communication (e.g. cache coherence), etc., which may consume even more energy than the energy saved by parallelization.

Another solution is to find one or more “average” speeds to execute the algorithm

such that the power profile is smoothed by lowering the height of the power spike; meanwhile, the energy will be also reduced. In an "ideal" solution, the minimum energy is achieved by running the processor at a constant speed such that the task is finished just before the end of the allocated time slot. However, the value of this optimal speed must be determined at the beginning to process the task, but it cannot be known until the last instruction (actually the last branch instruction) of the algorithm is completed. Thus, such an "ideal" solution is generally not possible. A partial solution is to partition the algorithm into several segments such that an average speed can be found for each segment. The energy spent within the segment is reduced; and it also helps to lower the power spike.

5.4 Super DVS: Energy Efficiency through Parallelism

We present our new *super DVS* technique in Section 5.4.1 to improve the energy efficiency of the ATR algorithm. It explores parallelism of the algorithm by first partitioning the code, and then pipelining the execution of all partitions. This allows to distribute the workload to multiple processors running at slower speeds. For each processor, we apply DVS and derive the optimal constant speed that minimizes energy and peak power at the same time. The new technique is called super DVS in the sense that DVS is applied to each small partition of the code. Parallel execution normally needs special treatment for communication. We propose a simple data handling technique in Section 5.4.2 by managing FIFO buffers between processors. In Section 5.4.3, we propose an additional scheme that processes multiple frames together to further reduce energy consumption.

5.4.1 Super DVS

We first partition the algorithm into a few independent segments such that the average speed can be found for each segment. Then, the partitioned code can be pipelined in

a multi-processor architecture. Finally, DVS is applied to each processor to achieve energy and power reduction.

Partitioning: finding a constant speed for each partition

With intra-task DVS, the code of the whole algorithm is divided into segments that are executed at different speeds. It is sometimes preferable to find a constant speed to reduce energy consumption. Although such a goal is normally difficult for the entire algorithm, finding the average speed for a portion of the algorithm is possible. We observe that such possibility is largely dependent on the way in which the algorithm is composed.

When intra-task DVS is applied to the ATR algorithm, due to the frequency changes at A and B, many different speeds will be applied to run the algorithm. Even the same code segment can be executed at different speeds. For example, if there are 5 targets detected for a given frame, module FFT will be executed 5 times, possibly at different speeds. However, because there is no loop-carried dependency across different iterations of the outer loop (as well as the inner loop), we can reorder the execution of the loops to execute those 5 instances of FFT together at a single speed. Similarly, the potential constant speeds can also be applied to IFFT and COMPUTE DISTANCE.

The ATR algorithm must be reconstructed to apply this technique. In the two-level nested loop, the iterations on both inner loop and the outer loop are executed to completion. Although this is the natural way to exercise loops, we have to partition the loop and reorder the sequence to access FFT, IFFT and COMPUTE DISTANCE for finding the average speed for each segment.

We reconstruct the algorithm and partition the two-level nested loop into three stages. In STAGE1, the algorithm finishes FFT for all M ROIs. Then, IFFT for M ROIs with different templates is performed in STAGE2. Finally, those K ROIs whose target distances need to be calculated are processed together in STAGE3. The modi-

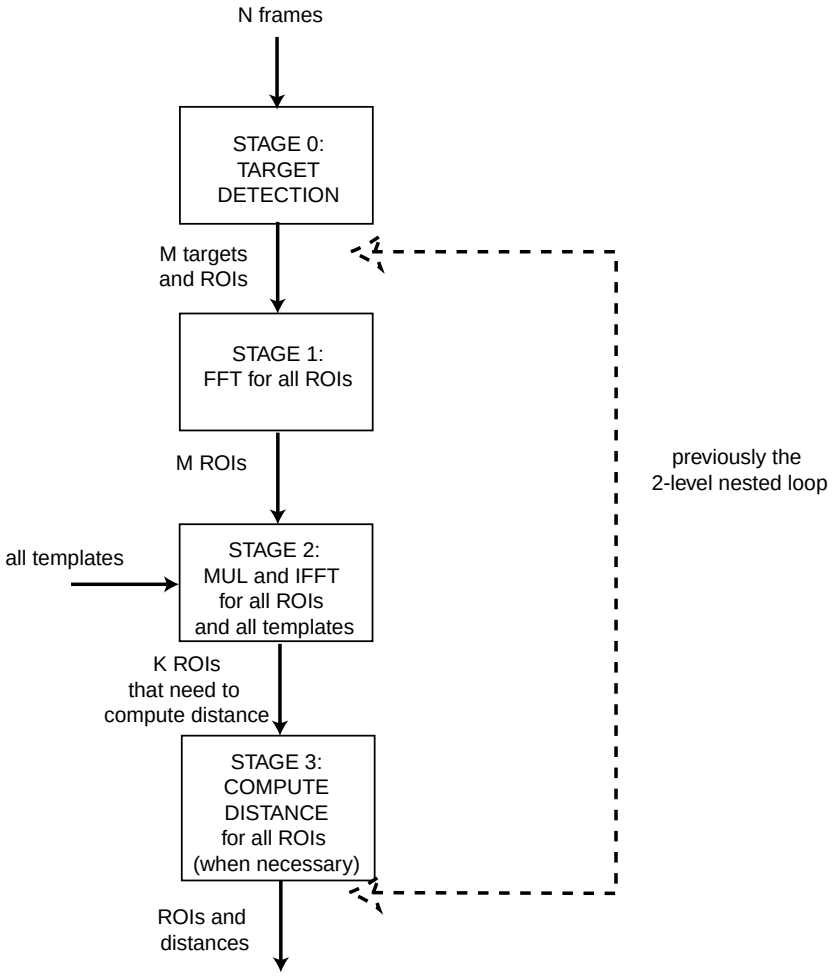


Figure 5.4: Partition the nested loop into stages.

fied block diagram is shown in Fig. 5.4. At the beginning, M targets are detected in STAGE0. Thereafter, the ROIs for these targets flow through the three following stages.

After the code transformation, the entire algorithm can be executed at four constant speeds (actually three speeds: STAGE2 and STAGE3 can be executed in the same speed since there is no conditional branch between them) for four stages. Special treatment must be applied to STAGE3 to achieve a constant speed. The *if* condition can be resolved at STAGE2 after IFFT by attaching some flags to each ROI indicating whether computing distance is needed. Therefore, the remaining workload $WCIC$ of STAGE3 ($WCIC(stage3) = K \times IC(COMPUTEDISTANCE)$) is known before it is executed such that the average speed can be calculated at the beginning of STAGE3.

Through analysis, we observed this code partitioning and transformation technique can moderately reduce energy by 5% - 15%. The reduction largely depends on the input data set and the size of each partition. The detailed discussion is outside the scope of this chapter. We still present the sketch of this technique because it naturally leads to a parallel version of the algorithm.

Parallelization through pipelining

Parallelization on the algorithm can be helpful for power management. For example, if an algorithm can be parallelized into two processors with each processing half workload and running at half speed, an approximately $4\times$ energy reduction will be achieved even after one new processor is added. However, in general it is quite difficult to parallelize an algorithm that has already been implemented in a serial style.

In the ATR algorithm, we observed that there is no data dependency between different image frames. Therefore, the code partition in Fig. 5.4 is ready to be executed in parallel, by pipelining the execution of all stages on different processors. After pipelining the algorithm, each stage itself will have the execution time of one entire frame delay, while all four stages together consume one frame delay previously. As a result,

more energy saving is available by slowing down the speeds of all processors. Ideally, if all four stages are perfectly balanced, that is, they have the same workload, the optimal energy reduction could be expected. However, this is generally not possible. For example, the workload of STAGE2 is about three times as much as that of STAGE1, since there will be three IFFTs for each FFT. Also, the workload of STAGE0 is fixed; while in other stages the workload will vary.

Super DVS: reducing energy in parallel

The final step is to apply DVS to each pipelined processor to improve energy efficiency. We call the new technique super DVS since it applies DVS to a smaller partition of the code. In Section 5.4.1 we already constructed the code partition for each stage such that the code can be executed at a constant speed. In Section 5.4.1 we extended the time slot for each stage as one frame delay for all processors. Therefore, we can find an optimal speed for each stage that maximally utilizes the extended time slot.

Fig. 5.5 describes the parallel super DVS algorithms for four pipeline stages, respectively. At the beginning of each stage, the processor speed is set according to the *WCIC* to be executed, which is always known ahead of time. Therefore, each processor can set a near-optimal speed such that the execution of the code completes just in one frame delay.

1. For STAGE0, the workload is set to $WCIC(STAGE0)$. The actual $IC(STAGE0)$ is only slightly different from $WCIC(STAGE0)$ since the module TARGET DETECTION is most computation intensive, while the instructions in the small loop to handle buffers are trivial. Therefore, when the speed of its processor is decided by the frame delay *delay* and $WCIC(STAGE0)$, the time slot with a duration *delay* is almost fully utilized. The speed to execute STAGE0 fixed for all frames if the same *delay* is given. Stage0 computes the number of targets in variable *M* and put *M* ROIs into buffer *ROI*0.

```
STAGE0(time delay, image im, buf ROI0)
    setProcessorSpeed(delay/WCIC(STAGE0))
    TARGET := TARGET DETECTION(im)
    M := 0
    for each (target ∈ TARGETS) loop
        ENQUEUE(ROI0, getRoi(im,target))
        M := M + 1
    end loop
    return M

STAGE1(time delay, buf ROI0, buf ROI1, int M)
    setProcessorSpeed(delay/(M ∗ WCIC(L1)))
    for i := 1,M loop L1
        roi := DEQUEUE(ROI0)
        ENQUEUE(ROI1, FFT(roi))
L1: end loop
    return M

STAGE2(time delay, buf ROI1, buf ROI2, int M,
            array TEMPLATE, float Val_th)
    setProcessorSpeed(delay/(M ∗ size(TEMPLATE) ∗ WCIC(L2)))
    K = 0
    for i = 1,M loop
        roi0 := DEQUEUE(ROI1)
        for each (template ∈ TEMPLATEs) loop L2
            roi1 = IFFT(roi0 × template)
            if roi1.Val > Val_th then
                K := K + 1
                ENQUEUE(ROI2,roi1)
            end if L2end loop
    end loop
    return K

STAGE3(time delay, buf ROI2, buf ROI3, int K)
    setProcessorSpeed(delay/(K ∗ WCIC(L3)))
    for  i = 1,K loop L3
        roi := DEQUEUE(ROI2)
        roi.distance := COMPUTE DISTANCE(roi)
        ENQUEUE(ROI3,roi)
L3: end loop
    return K
```

Figure 5.5: Parallel algorithms for super DVS.

2. For STAGE1, the processor speed is set according to M, which is the number of ROIs in buffer $ROI0$, $delay$, and the workload in loop L1 $WCIC(\mathrm{L1})$. This is because loop L1 will be executed M times during one frame delay, and the $WCIC$ of this loop can be analyzed statically. STAGE1 generates M ROIs in space-frequency domain after FFT, and puts them into buffer $ROI1$.

3. In STAGE2, the processor speed is set based on the parameters M, $delay$, $WCIC(L2)$, and $size(TEMPLATE)$, since STAGE2 will execute loop L2 in $M * size(TEMPLATE)$ times. Although there is an *if* block inside loop L2, the variance is trivial compared with computation-intensive IFFT. Therefore, using $WCIC(L2)$ to compute the average speed is almost optimal. This stage produces K ROIs to buffer $ROI2$, based on whether it is necessary to compute their distances.

4. Finally, in STAGE3, the processor speed is decided by K, $delay$ and $WCIC(L3)$. K ROIs are extracted from the input buffer $ROI2$ and distances are computed accordingly in loop L3. The results are queued to an output buffer $ROI3$.

The power profile for super DVS is shown in Fig. 5.6. During each period of one frame delay, each processor is running at a constant speed. Therefore, the overall power profile of all four processors is also constant during each period. Super DVS produces a much smoother power profile compared to intra-task DVS by eliminating the high peak spike. Significant savings on both power and energy are achieved by running the new parallel algorithms.

5.4.2 Implementation related issues: buffer management

In this section we discuss a few implementation related issues. As the parallel algorithm shown in Fig. 5.5, the four parallel processors communicate via FIFO buffers containing data to be produced and consumed. Our assumption is a shared memory organization, shown in Fig. 5.7.

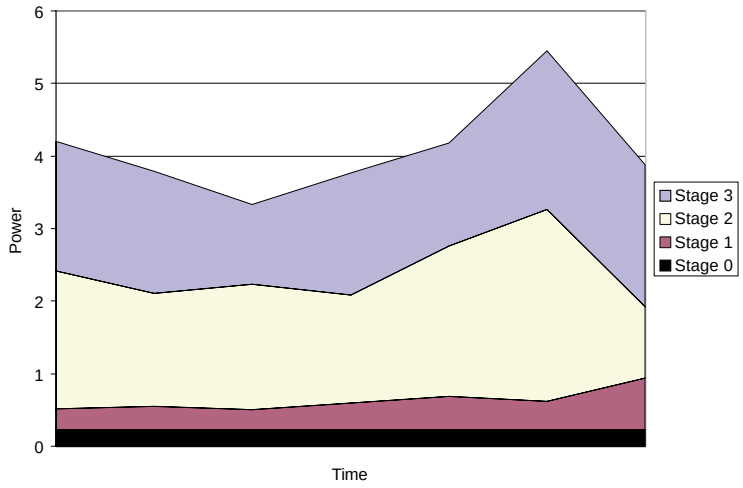


Figure 5.6: Power profile of super DVS.

In this memory organization, each stage produces new ROIs and appends them to the tail of its output buffer, which is the same as the input buffer of the next stage. Because the buffer size to be consumed next is passed to the consumer stage as the boundary of the input buffer, when the consumer stage fetches data from the head of the buffer, it will not access the data that is currently being generated by the producer. The variables to indicate workload M and K can be directly passed through to the next processor, or they can be organized in some other buffers in the same manner. The variable $delay$ is a constant or it can be set by external events, e.g. a user interface. It also can be accessed in separate FIFO buffers to avoid simultaneous access.

This buffer management scheme significantly simplifies the communication between parallel processors. Fig. 5.7 shows that, at any moment during the execution of this multi-processor system, any data in ROI buffers is accessed exclusively by at most one processor. Simultaneous accesses to the same data from two or more processors will never happen. (If variables $M, K, delay$ are also organized by separate buffers, there is no simultaneous access to these variables as well.) This suggest the data access is simplified to an extent that the processor does not need to acquire and release locks

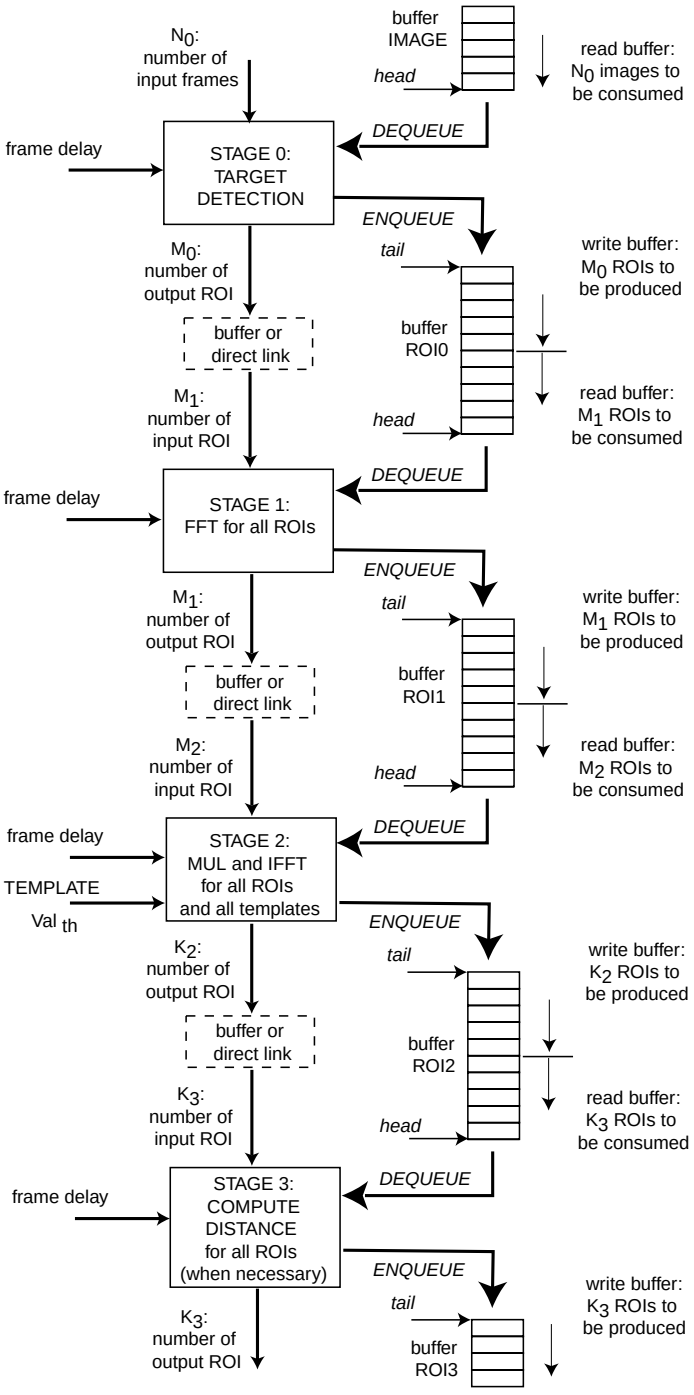


Figure 5.7: Pipelined processors with shared memory buffers.

```
STAGE0(time delay, buf IMAGE, buf ROI0, int N)
    setProcessorSpeed(delay/(N * WCIC(L0)))
    M := 0
    for i := 1,N loop
        im = DEQUEUE(IMAGE)
        TARGET := TARGET DETECTION(im)
        for each (target ∈ TARGET) loop L0
            ENQUEUE(ROI0, getRoi(im,target)
            M := M + 1
L0:     end loop
    end loop
    return M
```

Figure 5.8: Modified STAGE0 to process N frames at a time.

to access critical sections, since their is no critical section at all. For the same reasons, the cache coherence protocol is not necessary to implement such a system, if each write to the buffers will always commit to the shared memory. In fact, in many embedded DSP applications, the processor may not have caches.

By this simple buffer managing technique, we can avoid some sophisticated data handling mechanisms in general-purpose multi-processor systems, since they are not necessary in this specific application. The implementation of such a system is also easier than a general-purpose multi-processor system.

Although this parallel solution can reduce the energy of the processors, it may increase the energy in the memory system. For example, the memory must be designed to be multi-ported, which will increase the energy consumption. As many DVS studies do not consider the impact ofco the memory system, we also focus on the energy reduction to the processors.

DVS control is made very easy for this multi-processor system. Since all processors are working on totally independent data sets, the DVS control of each processor is independent to each other, as far as they all finish in allocated time slots with the same duration $delay$. In the shared memory organization, the four running processors with different speeds pose some challenges to the memory designer. The memory system

must accommodate simultaneous accesses from processors at different speeds.

One alternative solution is to use some processors that can directly pass its produced data to the consuming processor. Each processor has built-in input and output buffers to hold its local data set. The entire content of the buffer can be read/written from/to other processors. If applicable, this type of processor is an appropriate choice to implement ATR algorithm with super DVS technique. A sketch of directly linked data communication is shown in Fig. 5.9

5.4.3 Coarser granularity: processing multiple frames together

Our super DVS scheme reduces the energy and power of the processors by introducing a 3-frame delay for each frame. If more delay is allowed, we have an additional scheme that can save even more energy.

We have discussed previously that it is preferred to find an average processor speed at a coarser granularity (more workload). We have already applied this technique in Section 5.4.1. We observed that different frames will be processed at different speeds in STAGE1, STAGE2 and STAGE3 (STAGE0 always has the same speed), if we can find out the average speeds to process multiple frames, we can expect more energy reduction.

Such an addition is a straight-forward extension to the existing scheme. The only change is that STAGE0 will now fetch N frames from its input buffer, which contains multiple frames; and the *delay* will be changed accordingly by N times for N frames. (Both N and updated *delay* are set externally.) The delay for each frame is now ranging from $3N$ to $4N-1$ frame delay. The modified algorithm for STAGE0 is shown in Fig. 5.8. No change to the other stages is necessary.

In this chapter we present two types of techniques: I. parallelization (Section 5.4.1), and II. averaging the processor speed at a coarser grain (Section 5.4.1, 5.4.3). Our new super DVS technique is a combination of these two techniques, plus DVS. It is notable

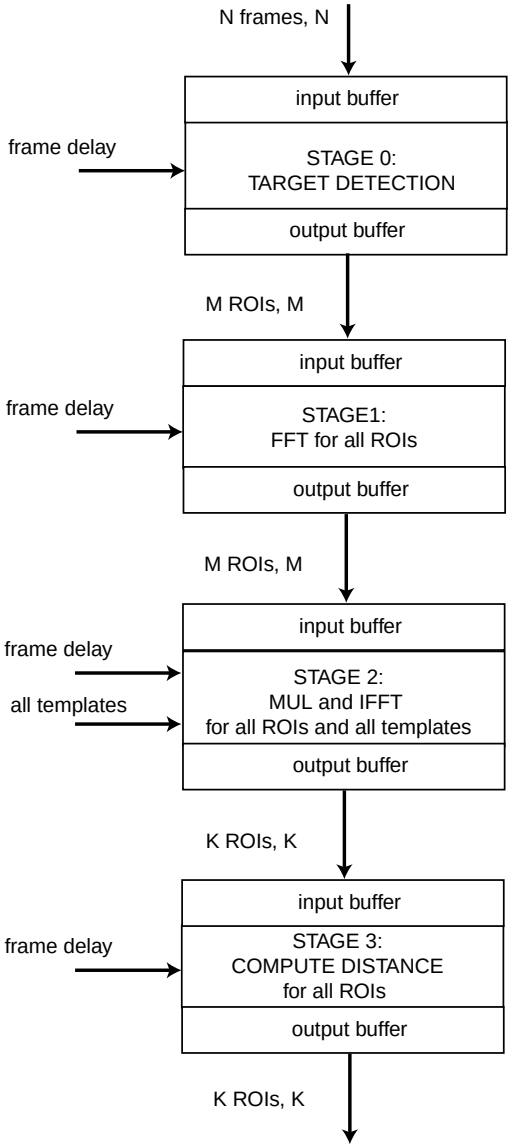


Figure 5.9: Pipelined ATR with directly linked data connection.

Parameter	*Description*	*Value*
Vmax	the maximum supply voltage	3.3V
Vth	the threshold voltage	0.8V
Fmax	the maximum frequency	1GHz
Pmax	the maximum power consumption	normalized to 100 when Nsw = 1
Tmax	the time overhead to swith between on and off	50 μs, 50k cycles at full speed
V, F, P,	current voltage, frequency and power	to be calculated

Table 5.1: An abstract model of a voltage-scalable processor.

that these new techniques and the existing power management techniques, including DVS (intra-task, and inter-task), static voltage scaling (SVS), shutting down idle components, and etc., are orthogonal techniques such that they can be applied either individually, or in combination. For example, if another parallel partition is available without finding an average speed for each processor, power and energy savings are still available. If DVS is not allowed, SVS can be applied to pipelined processors.

We will present some analytical results on energy reduction of super DVS technique in next section.

5.5 Analytical Results on Energy Reduction

In this section we present an analytical study to the new super DVS technique. We compare the results with the best-known existing technique, which is intra-task DVS.

5.5.1 An empirical processor model

We assume an abstract processor model for our analysis. A parameterized voltage-scalable processor is shown in Table 5.1. The peak power value is normalized to 100 to simplify the comparison. We also assume all the four processors running pipelined ATR algorithms with super DVS are the same type of the processor as the one which is running the serial algorithm with intra-task DVS, although in practical concerns they will probably be different types of processors.

Segment	WCIC (1000 instruction)	Nsw
L0	400	1
L1	170	1
L2	194	1
L3	377	1

(a) parameters of code segments

Series	Numbe of Frames	Numbe of targets per frame	Frame delay
A	200	0 - 2 (light workload)	16.7ms
B	200	5 - 7 (heavy workload)	16.7ms
C	200	random	16.7ms

(b) parameters of input data series

Table 5.2: Parameters of the code and input data.

We use the following equations to compute parameter P, F, V.

$$F = C_f \frac{(V - V_{th})^2}{V} \tag{5.1}$$

$$P = C_p N_{sw} V^2 F \tag{5.2}$$

C_f and C_p are processor dependent constants. They can be computed by applying maximum values of V, F, P. N_{sw} is a factor indicating the number of switching activities per cycle, it depends on the program.

5.5.2 Properties of the algorithm and data set

Table 5.2(a) gives the instruction count of the loops in each stage. These numbers are used in both intra-task DVS and super DVS to update the workload. The four code segments do not show a large variance in power consumption, indicating that the factor N_{sw} is about the same for all segments. For simplicity we set them all to 1. We assume the frame rate is 60 frames per second. Therefore the frame delay is 16.7ms.

We apply three series of input images. The first series A has very few targets in each frame, ranging from 0-2. That is, the workload of the algorithm is far less than the worst case. The second series B has a heavy workload with 5-7 targets per frame. In set C, the number of targets is random. The input data sets are summarized in Table 5.2(b)

Technique	Energy		Peak Power	
	value *	% reduction	value *	% reduction
Intra-task DVS	11.7		63.7	
Super DVS (N =1 frame)	4.53	61.3%	3.17	95.0%
Super DVS (N = 2 frames)	4.45	62.0%	3.17	95.0%
Super DVS (N = 4 frames)	4.38	62.6%	2.89	95.5%
Super DVS (N = 8 frames)	4.34	62.9%	2.10	96.7%

Input image data: series A (light workload)

Technique	Energy		Peak Power	
	value *	% reduction	value *	% reduction
Intra-task DVS	84.4		63.7	
Super DVS (N =1 frame)	33.4	60.4%	14.9	76.6%
Super DVS (N = 2 frames)	33.2	60.7%	12.8	80.0%
Super DVS (N = 4 frames)	33.1	60.8%	11.3	82.3%
Super DVS (N = 8 frames)	33.1	60.8%	10.7	83.2%

Input image data: series B (heavy workload)

Technique	Energy		Peak Power	
	value *	% reduction	value *	% reduction
Intra-task DVS	49.5		63.7	
Super DVS (N = 1 frame)	19.7	60.2%	18.8	70.5%
Super DVS (N = 2 frames)	17.9	63.8%	11.9	81.3%
Super DVS (N = 4 frames)	17.0	65.7%	10.9	82.9%
Super DVS (N = 8 frames)	16.4	66.9%	9.46	85.1%

Input image data: series C (random workload)

* energy and power values are normalized

Table 5.3: Energy and power saving achieved by super DVS.

5.5.3 Power and energy reduction by super DVS

Table 5.3 shows both energy and (peak) power reduction of super DVS compared with intra-task DVS. We also vary the number of frames per image group to examine the effect of an increased granularity by averaging the larger workload among multiple frames. In all three input series, super DVS exhibits a 60% energy reduction;and the peak power can be reduced by as much as 80%-90%. We make following observations from the analytical results.

1. Intra-task DVS is not quite “low-power” in the sense that its peak power is always

the same (the high-power spike) regardless of the workload, because it always starts from the worst case. On the other hand, super DVS adapts both energy and power to the workload very well.

2. Super DVS can significantly reduce both power and energy compared with intra-task DVS. The percentage reduction to peak power is more than the saving on energy. This is because super DVS produces a more smoothed power profile, while the power profile of intra-task DVS always has high peaks and sharp jitters.

3. When the workload does not vary too much (series A and B), processing multiple frames together may not yield extra gains. This is due to the fact that the average speed in multiple frames is not quite different from speeds in individual frames. While in series C, where the workload varies in a diverse range, additional savings can be achieved at a coarser granularity.

5.5.4 The impact of DVS overhead

Not every DVS study has considered the DVS overhead, the overhead to change the voltage of the processor. Mostly they assume it is either free, that is, the frequency and voltage of the processor can be changed in zero time and zero energy, or the overhead is trivial. These assumptions are generally not true.

We propose an abstract model for an analytical study to see how much the overhead can impact the energy cost of DVS techniques. Our model is simple: we assume a parameter T_{max}, which is the maximum time overhead when the processor is switched between off ($F = 0$) and full speed F_{max}. T_{max} is a processor dependent constant. It indicates how quickly the processor can switch from one voltage/frequency setting to another. We assume $T_{max} = 50\mu s$, equivalent to 50,000 cycles at maximum clock speed.

For a frequency/voltage change from setting V_1/F_1 to V_2/F_2, the time overhead is scaled by the frequency difference F_2 and F_1, that is,

DVS granularity / *Techniques*	*Coarse-grain 100% workload 100% frame delay*	*Fine-grain 50% workload 50% frame delay*	*Very-fine-grain 10% workload 10% frame delay*
Intra-task DVS	3.87%	7.52%	29.5%
Super DVS (N = 1 frame)	0.002%	0.004%	0.02%
Super DVS (N = 2 frames)	0.001%	0.001%	0.01%
Super DVS (N = 4 frames)	<0.001%	0.001%	0.005%
Super DVS (N = 8 frames)	<0.001%	<0.001%	0.003%

Table 5.4: Energy overhead vs. different DVS granularity.

$$T_{V_1/F_1-V_2/F_2} = T_{max}\frac{|F_2 - F_1|}{F_{max}} \tag{5.3}$$

For simplicity, we also assume during the switching time, the average power overhead is the average of P_1 and P_2, which refer to the power consumption before and after the change. The energy overhead can be computed accordingly. In reality, the energy spent on DVS may be even higher. Some other models of the overhead to change processor voltage can be found in [13, 28].

We present the impact of voltage-changing overhead to intra-task DVS and super DVS in Table 5.4. Data series A is applied for this study. The first column of the results shows that intra-task DVS spends up to 4% of energy on changing voltage, which may not be considered as "trivial"; while such overhead for super DVS is less than trivial. We could try to increase T_{max} and see how sharply the overhead grows on intra-task DVS. However, because it may not be meaningful to have very large values of T_{max} for recent processors, we consider the equivalent cases by working on a smaller data set while fixing T_{max}. This refers to fine-grained DVS, in which cases the voltage changes are made quite frequently such that the DVS overhead is comparable to the slack time being saved by DVS.

In the ATR algorithm, suppose we want the algorithm to work on images with smaller size or less pixels. Therefore, the *WCIC* of the program segments in Table 5.2 will all decrease. We assume they will decrease by the same factor, approximately.

On the other hand, since the algorithm is dealing with less amount of data, its frame delay should be also decreased accordingly such that the performance of the algorithm is sustained (it is still processing the same amount of data in unit time). For example, if the image size is reduced by half (50% pixels), all the *WCIC* in Table 5.2 should reduce by a factor of 50%, while the frame delay also reduces to its half. This is a reasonable assumption for fine-grained DVS.

In Table 5.4 these cases are studied to examine the impact of overhead when DVS is applied in finer granularities. It indicates that intra-task DVS may not be scalable very well to finer granularities with the sharply growing energy overhead. For example, in the third column, intra-task DVS will spend 30% of the energy on changing voltage. This is because intra-task DVS potentially have more frequent and more dramatic voltage changes. Especially, during each frame the processor must be switched from low-power to high-power once with larger time and energy overhead; and this overhead will be even more expensive in fine-grained cases. In contrast, the DVS overhead still remains trivial for super DVS. Super DVS incurs four voltage changes per frame, and these changes are quite smooth with small overhead. The overhead can be further amortized by processing multiple frames together where only four voltage changes are applied to N frames. As a result, intra-task DVS (and potentially inter-task DVS) may not be scalable to finer-grained applications, and will suffer more from the overhead on slowly-switched processors. On the other hand, super DVS will be still applicable to finer-grained applications or on processors that switch between different voltage/frequency settings slowly.

5.6 Chapter Summary

Increasing the level of parallelism is known to be beneficial for power management. However, this area is often overlooked in recent studies due to the difficulty to parallelize serial algorithms. In this chapter we present a new technique that effectively

parallelizes an image processing algorithm such that it could be pipelined in a four-processor system. The energy efficiency is improved by slowing down each processor; meanwhile the performance requirement is compensated by increased parallelism. We propose a super DVS technique that combines the parallel approach with DVS. We successfully discover an optimal voltage setting for each processor to minimize both energy and peak power. The new technique will enable the existing DVS techniques to further reduce energy by the next order of magnitude. It also reduces the DVS overhead and is proven to be more scalable to finer-grained applications, where the overhead to change the processor's voltage setting is very costly. We believe our new technique is generally applicable to a large class of signal processing applications with regular data access patterns. The work proposed in this chapter represents one of the core CAD tools in a larger design framework for energy efficient embedded systems.

Chapter 6

Communication Speed Selection and Partitioning

High-speed serial network interfaces are becoming the primary way for modern embedded systems and systems-on-chip to connect with each other and with peripheral devices. Modern communication interfaces are capable of operating at multiple speeds and are opening a new dimension of trade-offs between computation and communication. Unfortunately, today's CPU-centric techniques often fail to consider multi-speed communication and the balance between communication and computation for time and energy; as a result, they yield sub-optimal if not incorrect designs.

This chapter presents a new technique for global energy optimization through coordinated functional partitioning and speed selection for the processors and their communication interfaces. We propose a multi-dimensional dynamic programming formulation for energy-optimal functional partitioning with CPU/communication speed selection for a class of data-regular applications under performance constraints. We demonstrate the effectiveness of our optimization techniques with an image processing application mapped onto a multi-processor architecture with a multi-speed Ethernet.

6.1 Introduction

Towards High-Speed Serial Busses on SoC

A key trend in systems-on-chip is towards the use of high-speed serial busses for system-level interconnect. Serial busses offer many compelling advantages, including modularity, composability, scalability, form factor, and power efficiency [11, 40, 57].

Modularity and composability are extremely important, because the sheer complexity of these chips forces designers to raise the level of abstraction. Most SoC designs are done by integration of intellectual property (IP) components as a way to manage complexity while meeting time-to-market deadlines. Serial protocols are well understood and have long been used in in automotive control (e.g., CAN from Bausch) and consumer electronics (e.g., I^2C from Philips). More recent protocols such as FireWire (IEEE 1394) and USB are commonly used not only for peripheral devices but also for connecting multiple embedded processors. They provide a simple, standardized, efficient, and scalable way of building loosely coupled systems. High-speed serial controllers such as Ethernet are now an integral part of many embedded processors.

Serial busses also have power and form factor advantages. From automobiles to computer peripherals, serial interconnects such as FireWire and USB are compact and low power compared to SCSI or parallel, which are bulky, high power, and limited in length. This is especially important for systems-on-chip, where gates are virtually free, but wires are the most expensive part of the chip real estate. Long, parallel, shared wires are not only high power but also suffer from clock skews and even cross talks as the feature size shrinks. Serial controllers provide a clean abstraction by shielding components from these low-level concerns. Moreover, modern protocols also support plug-and-play and power management features such as subnet shutdown or link suspension. These features and more make high-speed serial protocols an attractive choice for rapid integration of SoC architectures.

Power/Performance Issues with Serial Networks

Of course, serial controllers come at a price. The area and IP licensing will have a cost, but this cost might be justified by time-to-market or other overriding business concerns. In fact, it might be even less of an issue for future IP, which will likely have these serial controllers integrated. For example, AMD's newly announced Au1100 [1] is a MIPS based microcontroller with integrated 10/100-base T Ethernet, USB, and many other I/O. However, power and performance will become the critical issues, as they directly affect the correctness of the design.

For power optimization, previous efforts focused on the processor for several reasons. The CPU was the main consumer of power, and it also offered the most options for power management, including voltage scaling. However, recent advances in both processors and communication interfaces are driving a shift in how power should be managed.

CPU-centric power management has given rise to a new generation of processors with dramatically improved power efficiency, and the CPU is now drawing a smaller percentage of the overall system power. The insatiable demand for bandwidth has also resulted in high-speed communication interfaces. Even though their power efficiency (i.e., energy per bit transmitted) has also been improved, communication power now matches or surpasses the CPU, and is thus a larger fraction of the system power. For instance, the Intel XScale processor consumes 1.6W at full speed, while a GigaBit Ethernet interface consumes 6W.

System Power Management with Speed Selection

Many communication interfaces today support multiple data rates. However, the scaling effects tend to be opposite those of voltage scalable CPUs. For CPUs, slower speed generally means lower power and lower energy per instruction; but for communication, faster speed means higher power but often less energy per bit. This is highly dependent

on the specific controller. Few research works to date explored communication speed as a key parameter for power optimization.

Speed selection cannot be performed for just communication or computation in isolation, because a local decision can have a global impact. One reason is that communication now goes through a shared medium rather than point-to-point. The CPUs cannot all be run at the slowest, most power-efficient speeds, because they must compete for the available time and power with each other and with the communication interfaces. A faster communication speed, even at a higher energy-per-bit, can save energy by creating opportunities for subsystem shutdown or voltage scaling the processors. Greedily saving communication power may actually result in higher overall energy. At the same time, functional partitioning must be an integral part of the optimization loop, because different partitioning schemes can dramatically alter the communication payload and computation workload for each node.

Approach

For a given workload on a networked architecture, our problem statement is to generate a functional partitioning scheme and to select the speeds of communication interfaces and processors, such that the total energy is minimized. In general, such a problem is extremely difficult. Fortunately, for a class of systems with pipelined tasks under an overall latency constraint, efficient, exact solutions exist. This chapter presents a multi-dimensional dynamic programming solution to such a problem. It formulates the energy consumed by the processors and communication interfaces with their power/speed scaling factors within their available time budget. We demonstrate the effectiveness of this technique with an image processing algorithm mapped onto a multi-processor architecture interconnected by a GigaBit Ethernet. This technique is also applicable as a heuristic to general dataflow problems.

6.2 Related Work

Previous works have explored communication synthesis and optimization in distributed multi-processor systems. [72] presents communication scheduling to work with rate-monotonic tasks, while [23] assumes the more deterministic time-triggered protocol (TTP). [49] distributes timing constraints on communication among segments through priority assignment on serial busses (such as control-area network) and customization of device drivers. While these assume a bus or a network protocol, LYCOS [37] integrates the ability to select among several communication protocols (with different delays, data sizes, burstiness) into the main partitioning loop. Although these and many other works can be extended to SoC architectures, they do not specifically optimize for energy minimization by exploiting the processors' voltage scaling capabilities.

Related techniques that optimize for power consumption of processors typically assume a fixed communication data rate. [10] uses simulated heating search strategies to find low-power design points for voltage scalable embedded processors. [45] performs battery-aware task post-scheduling for distributed, voltage-scalable processors by moving tasks to smooth the power profile. [69, 68] propose partitioning the computation onto a multi-processor architecture that consumes significantly less power than a single processor. [17] reduces switching activities of both functional units and communication links by partitioning tasks onto a multi-chip architecture; while [29] maximizes the opportunity to shut down idle processors through functional partitioning. All these techniques focus on the computational aspect without exploring the speed/power scalability of the communication interfaces.

Existing techniques cannot be readily combined to explore many timing/power trade-offs between computation and communication. The quadratic voltage scaling properties for CPU's do not generalize to communication interfaces. Even if they do, these techniques have not considered the partitioning of power and timing budgets among computation/communication components across the network. Selecting

communication attributes by only considering deadlines without power will lead to unexpected, often incorrect results at the system level.

6.3 System Model

This section defines a system-level performance/energy model for both computation and communication components in a networked on-chip multi-processor architecture. In this chapter, a system consists of M processing nodes $N_i, i = 1,2,\ldots,M$ connected by a shared communication medium. Each *processing node* (or *node* for short) consists of a processor, a local memory, and one or more communication interfaces that send and/or receive data from other nodes.

6.3.1 Jobs and Tasks

A *processing job* assigned to a node has three *tasks*: *RECV*, *PROC*, and *SEND*, which must be executed serially in that order. *RECV* and *SEND* are communication tasks on the interfaces, and *PROC* is a computation task on the processor. The *workload* for each task is defined as follows. For communication tasks *RECV* and *SEND*, workload W_r and W_s indicate the number of bits to be received and sent, respectively. For the computation task *PROC*, the workload W_p is the number of cycles. Let T_p, T_r, T_s denote the *delays* of tasks *PROC*, *RECV* and *SEND*, respectively. Let F_p denote the clock frequency of the processor, F_r and F_s the respective data bit rates for receiving and sending. We have

$$T_p = \frac{W_p}{F_p};\ \ T_r = \frac{W_r}{F_r};\ \ T_s = \frac{W_s}{F_s} \tag{6.1}$$

(6.1) is reasonable for processors executing data-dominated programs, where the total cycles W_p can be analyzed and bounded statically. However, it does not hold true

in general if the effective data rate can be reduced by collisions and errors on the shared communication medium. We present the collision-free condition of the shared medium in Section 6.4.

To model the non-ideal aspect of the medium, we introduce the *communication efficiency* terms, ρ_r and ρ_s,

$$0 \leq \rho_r, \rho_s \leq 1, \text{ such that } T_r = \frac{W_r}{\rho_r F_r} \text{ and } T_s = \frac{W_s}{\rho_s F_s}.$$

Note that ρ_r and ρ_s need not be constants, but may be functions of communication speeds F_r, F_s. For brevity, our experimental results assume an ideal communication medium ($\rho_r = \rho_s = 1$) without loss of generality. A more practical communication model can be directly applied, since ρ_r and ρ_s can be very well bounded for a collision-free medium.

D is a *deadline* on each processing job, which requires $T_r + T_p + T_s \leq D$ for the three serialized tasks. If any slack time exists, then we can slow down task *PROC* by voltage scaling to reduce energy. Therefore, we assume the job finishes at the deadline. That is,

$$D = T_r + T_p + T_s \tag{6.2}$$

6.3.2 Power Scaling

On each node, we assume only the processor and the communication interfaces are power-manageable by speed selection. The power consumption by the communication medium is interpreted to be the total power consumed by all active communication interfaces. We assume a processor's voltage-scaling characteristics can be expressed by a scaling function $Scale_p$ that maps the CPU frequency to its power level. A communication interface also has scaling functions that characterize the power levels at different communication data rates for sending and receiving. (6.2) implies $Scale_p$ is continu-

ous, while communication interfaces support only a few discrete scaling points. Let P_p, P_r, and P_s denote the power levels of tasks *PROC*, *RECV* and *SEND*, respectively, then,

$$P_p = Scale_p(F_p);\;\; P_r = Scale_r(F_r);\;\; P_s = Scale_s(F_s) \tag{6.3}$$

Let P_{ovh} denote the power overhead when introducing an additional node into the system. It captures the power of the memory, minimum power of the CPU and communication interface, CPU's power during *RECV* and *SEND* (DMA), and communication interfaces' power during *PROC*.

The *energy consumption of a task* is the power-delay product. Let E_p, E_r, E_s, and E_{ovh} denote the energy consumption of tasks *PROC*, *RECV*, *SEND*, and overhead of a node,

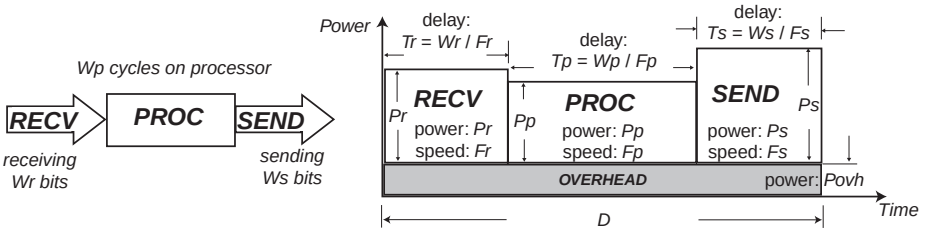
$$E_p = P_p T_p;\;\; E_r = P_r T_r;\;\; E_s = P_s T_s;\;\; E_{ovh} = P_{ovh} D \tag{6.4}$$

For one node N_i with tasks $PROC_i$, $RECV_i$, and $SEND_i$, the *total energy of node* N_i is

$$E_{N_i} = E_{p_i} + E_{r_i} + E_{s_i} + E_{ovh_i} \tag{6.5}$$

Fig. 6.1 shows the structure of a processing node. The gray bar represents the overhead and white bars represent tasks *RECV*, *PROC* and *SEND*. The area of the bars refers to the energy contribution of the tasks and overhead.

Finally, the *total energy of the system* is the sum of energy consumption on each node,



(a) block digram (b) timing-power digram

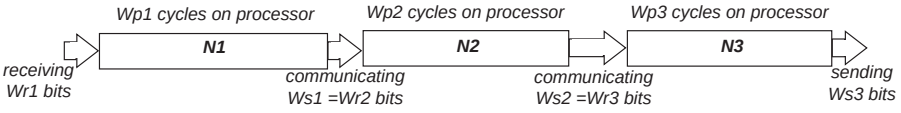
Figure 6.1: Timing and power properties of a processing node.

$$E_{sys} = \sum_{i=1}^{M} E_{N_i} \tag{6.6}$$

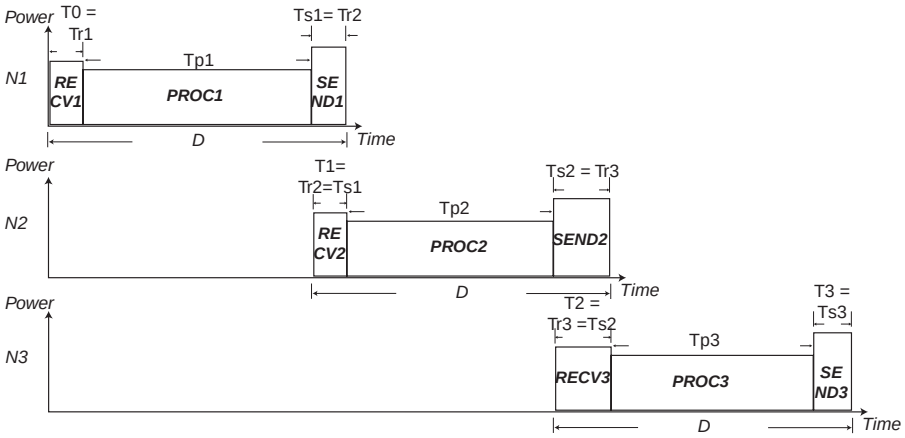
6.3.3 M-Node Pipeline

This chapter considers a special case called an *M-node pipeline*. It consists of identical nodes $N_i, i = 1,2,\ldots,M$ as characterized by $Scale_p, Scale_r, Scale_s, E_{ovh}$. Each node N_i receives W_{r_i} bits of data from the previous node N_{i-1} (except N_1), processes the data in W_{p_i} cycles, and sends the W_{s_i}-bit result to the next node N_{i+1} (except N_M). Each $SEND_i \rightarrow RECV_{i+1}$ communication pair sends and receives same amount of data at the same communication speed, with the same communication delay, and we assume they start and finish at the same time. That is, $W_{s_i} = W_{r_{i+1}}, F_{s_i} = F_{r_{i+1}}, T_{s_i} = T_{r_{i+1}}$. All nodes have the same deadline D, and each node acts as a pipeline stage with delay D. Fig. 6.2 shows an example of a three-node pipeline. For brevity, the overhead is not shown. Fig. 6.2(c) shows the pipelined timing diagram by folding the tasks in Fig. 6.2(b) into a common interval with duration D, which is the delay of each pipeline stage. During each time interval with a duration D, the first node of the pipeline will be fed with one set of incoming data; meanwhile one set of resulting data will be produced by the last node. Section 6.4 presents the schedulability conditions for an M-node pipeline based on collision and utilization of the shared communication medium.

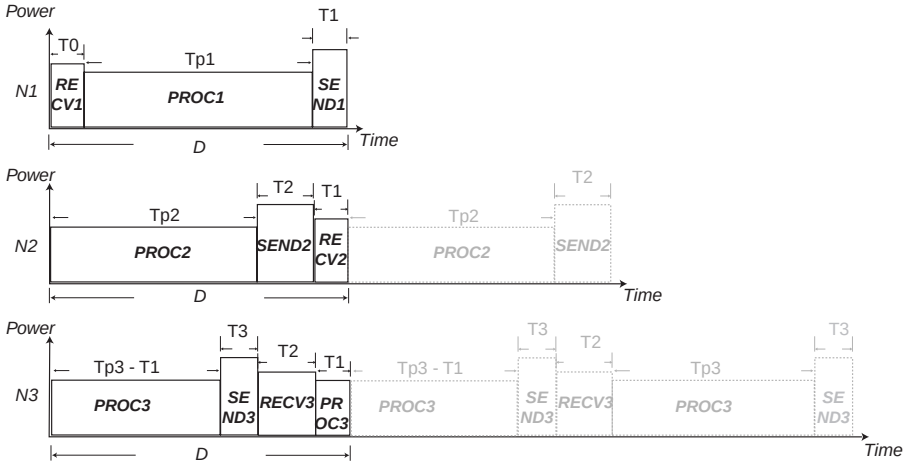
An M-node pipeline can be partitioned and mapped onto an M'-node pipeline ($M' \leq$



(a) block diagram



(b) serialized timing-power diagram



(c) pipelined timing-power diagram

Figure 6.2: A 3-node pipeline.

M) by merging adjacent nodes $N_i, N_{i+1}, \ldots, N_{i+j} (j \geq 1)$ into a new node N'_k. The new node N'_k combines all computation workload, receives W_{r_i} bits of data, and sends W_{s_j} bits of data. Communication within a node become local data accesses. That is, $W'_{p_k} = \sum_{l=0}^{j} W_{p_{i+l}}$, and $W'_{r_k} = W_{r_i}, W'_{s_k} = W_{s_j}$. The new M'-node pipeline is called a *partitioning* of the initial M-node pipeline.

6.4 Schedulability Conditions

This section presents the schedulability conditions for the pipelined on-chip multiprocessor system. In the pipelined timing diagram Fig. 6.2(c) of the three-node pipeline, we fold the tasks in Fig. 6.2(b) into a common interval with duration D, which is the delay of each pipeline stage. Note that there appear to be two instances of task *PROC* on node N_3. This does not mean that task *PROC* on node N_3 is preempted. In fact, each instance is a part of an integrated task *PROC* across the boundary between pipeline stages. In other words, the boundary between pipeline stages resides in the middle during the execution of task *PROC*.

Fig. 6.2(c) shows that due to the common deadline D, communication activities are shifted to different time slots, such that at any given time, there is at most one active communication instance (a $SEND_i \rightarrow RECV_{i+1}$ pair, e.g. $SEND2 \rightarrow RECV3$ and $SEND1 \rightarrow RECV2$ are serialized). This is especially meaningful if all nodes share the communication medium such as Ethernet instead of point-to-point connections. If collision does not occur, then our estimation on both performance and energy of the whole system can be well bounded. Collision is always undesirable because retransmission costs both time and energy. Communication activities should be scheduled such that the system is collision-free.

Lemma 5 (Collision-free Condition) In an M-node pipeline with a deadline D, let

$T_i, i = 0, 1, \ldots, M$ indicate the delays of $M+1$ instances of data communication.

$$T_i = \begin{cases} T_{r_1} & (i = 0) \\ T_{s_i} = T_{r_{i+1}} & (i = 1, 2, \ldots, M-1) \\ T_{s_M} & (i = M) \end{cases}$$

The system does not have collision on the shared communication medium **iff** the *utilization* of the shared communication medium is less than or equal to 1. That is,

$$U = \sum_{i=0}^{M} \frac{T_i}{D} \leq 1 \tag{6.7}$$

Note that for a general multi-processor, Lemma 5 expresses the *overload* condition and can be only a necessary condition for a collision-free schedule. However, it is also a sufficient condition for M-node pipelines as defined in Section 6.3.3, because this special case of pipelining has the property of serializing all communication instances. Lemma 5 is also the schedulability condition for the shared communication medium.

Lemma 6 (Schedulability Condition of One Node) In an M-node pipeline with a deadline D, $\forall$ nodes $N_i, i = 1, 2, \ldots, M$, N_i is able to meet the deadline D **iff** N_i is not *overloaded*, that is,

$$\frac{W_{p_i}}{\max(F_{p_i})} \leq D - T_{r_i} - T_{s_i} \tag{6.8}$$

Lemma 6 states the *overload* condition of one node: given the communication speeds (that determine communication delays T_{r_i}, T_{s_i}), if its computation task cannot be completed before the time budget $D - T_{r_i} - T_{s_i}$ by operating at the maximum CPU clock rate, then this node will fail to meet the deadline D and thus the whole pipeline will be malfunctioning. If Lemma 6 cannot be satisfied, then the only way to meet the deadline is to select higher communication speeds to reduce T_{r_i}, T_{s_i}, in order to allocate

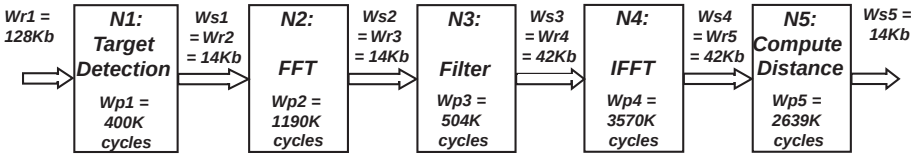


Figure 6.3: Functional blocks of the ATR algorithm.

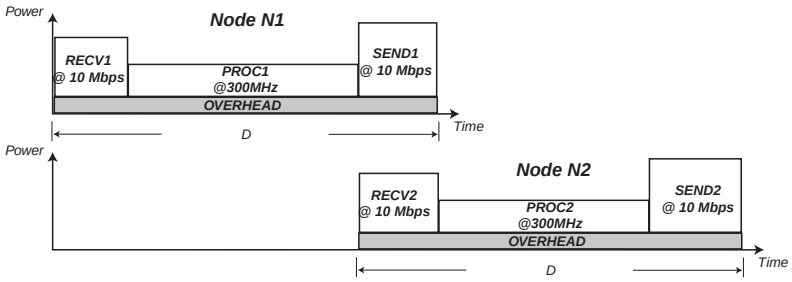
additional time budget for computation. High-speed communication can also reduce communication collision to satisfy Lemma 5.

Lemma 7 (Schedulability Condition of the System) An M-node pipeline is schedulable to meet a deadline D **iff**
(1) $\forall$ node $N_i, i = 1, 2, \ldots, M$, N_i meets the deadline D (Lemma 6), and (2) The shared communication medium is collision-free (Lemma 5).

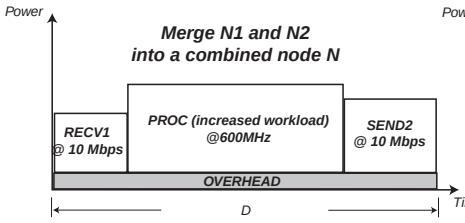
Lemma 7 says that the system's schedulability is determined by the schedulability of all resources, including M nodes and the communication medium. If and only if none of them is overloaded, the system can be pipelined by the deadline D. Lemma 7 holds true only for this M-pipeline organization; it is a necessary but not sufficient condition for a general multi-processor system.

6.5 Motivating Example

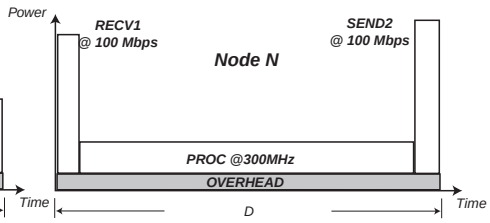
We use an automatic target recognition (ATR) algorithm (Fig. 6.3) as our motivating example. Originally it is a serial algorithm. We reconstructed a parallel version and mapped it onto pipelined multiple processors. Pipelining allows each processor to run at a much slower speed with a lower voltage level to reduce overall computation energy, while parallelism compensates for the performance. Of course, a multi-processor platform incurs energy for inter-processor communication, extra processors, memory, and other overhead.



(a) A fine-grain partitioning scheme reduces energy on computation, at the cost of inter-proessor communication and overhead of additional nodes.



(b) The combined node reduces communication and overhead, but it requires more energy for computation.



(c) The computation energy can be reduced by increasing communication speeds, which leaves more time on computation.

Figure 6.4: The impact of different partitioning schemes and communication speed settings.

Mapping Task to Node through Partitioning

Given the five functional blocks (tasks) of the ATR algorithm, several partitioning schemes are possible for mapping the tasks to a number of pipelined nodes. Fig. 6.4 shows an example by considering how they map the first two tasks onto nodes. In Fig. 6.4(a), they are mapped onto two nodes $N1$ and $N2$ that are both allowed to operate at a lower speed (300MHz) for computation. This scheme has lower computation energy than if they were mapped onto one node, but it requires energy on communication tasks $SEND1 \rightarrow RECV2$, plus overhead. Fig. 6.4(b) shows a mapping onto one node. It eliminates the communication $SEND1 \rightarrow RECV2$ and the overhead of an extra node. However, the combined node has much more computation workload and must run at a faster clock rate (600MHz), a less energy-efficient level.

Zooming out, many partitioning schemes are possible, even when limited to a pipelined organization. For example, one partitioning $[N1, N2][N3, N4, N5]$ may be optimal for nodes $N1$ and $N2$; but it will preclude another solution $[N1], [N2, N3], [N4, N5]$ that may lead to lower energy for the whole system.

Speed Selection for CPU and Communication

In additional to partitioning, the selection of communication speed is an equally critical issue. For example we consider a 10/100/1000Base-T Ethernet interface. It consumes more power than the CPU at high (100/1000Mbps) speeds, but less power than the CPU at the slower, 10Mbps data rate. In Fig. 6.4(b), the processor must operate at a high clock rate due to the low-speed communication at 10Mbps. Because of the deadline D, communication and computation compete for this budget. Low-speed communication leaves less time for computation, thereby forcing the processor to run faster to meet the deadline. Conversely, high-speed communication could free more time budget for computation, shown in Fig. 6.4(c), where the CPU's clock rate is dropped to 300MHz with 100Mbps communication. Although extra energy could be allocated to communi-

cation, if the energy saving on the CPU could compensate for this cost, then (c) would be more energy-efficient than (b).

The communication-computation interaction becomes more intricate in a multi-processor environment. Any data dependency between different nodes must involve their communication interfaces. The communication speed of a sender will not only determine the receiver's communication speed but also influence the choice of the receiver's computation speed. The communication speed on the first node of the pipeline will have a chain effect on all other nodes in the system. A locally optimal speed for the first node will not necessarily lead to a globally optimal solution.

Combining Partitioning and Speed Selection

Partitioning and communication speed selection are mutually enabling each other. Given a fixed partitioning scheme, the designers can always find the corresponding optimal speed setting that minimizes energy for that scheme. However, energy-optimal speed selection for a partitioning is not necessarily optimal over all partitionings. Instead, partitioning and speed selection are mutually enabling. In this chapter, we take a multi-dimensional optimization approach that considers performance requirement, schedulability, load balancing, communication-computation trade-offs, and multi-processor overhead in a system-level context.

6.6 Problem Formulation

Given an M-node pipeline, the choices of partitioning and communication speed settings will lead to different energy consumption at the system level. This section formulates the energy minimization problems by means of partitioning and communication speed selection. In both cases, the optimal solutions can be obtained by dynamic programming. Finally, the combined optimization problem with both partitioning and communication speed selection can be addressed synergistically by multi-dimensional

dynamic programming.

Problem 1 (Optimal Partitioning) Given

(a) M pipelined nodes N_i with workload $W_{p_i}, W_{r_i}, W_{s_i}$, $i = 1, 2, \ldots, M$,

(b) a deadline D for all nodes, and

(c) the constraint that the speed settings of all communication instance must match: $F_{r_1}, F_{s_i} = F_{r_{i+1}}, F_{s_M}$, for $i = 1, 2, \ldots, M-1$,

find a partitioning scheme that minimizes energy E_{sys}.

To avoid exhaustive enumeration in the $O(2^{M-1})$ solution space, we construct a series of sub-problems as follows. We consider a sub-problem $P[i, j]$ that maps the first j original nodes $N_1, N_2, \ldots, N_j$ onto a sub-partitioning i nodes $N'_1, N'_2, \ldots, N'_i$. The optimal solution of $P[i, j]$ has the minimum energy $E[i, j]$. It can be decomposed into two parts shown in Fig. 6.5: (a) a sub-partitioning $P[i-1, l]$ that maps first l original nodes to $i-1$ new nodes, plus (b) the i^{th} new node N'_i that combines the original nodes $N_{l+1}, \ldots, N_j$ with its energy denoted as $E_{N'i}$. In order to achieve the minimum energy $E[i, j]$, the energy consumption of (a) must also be an optimal sub-solution $E[i-1, l]$. Since l can be any value in a range $i-1 \leq l \leq j-1$, $E[i, j]$ must also be the minimum value of $E[i-1, l] + E_{N'_i}$ over all these possible values of l. That is, $E[i, j] = \min_{i-1 \leq l \leq j-1}\{E[i-1, l] + E_{N'_i}\}$. Any optimal sub-solution $E[i, j]$ can be derived from other optimal sub-solutions $E[i-1, l]$. Therefore, the problem has an *optimal sub-structure* and a *dynamic programming* approach is appropriate. It is illustrated in Fig. 6.6. Matrix $E[i, j]$ is initialized to ∞ for $0 \leq i \leq j \leq M$. We define $E[0, 0] = 0$ and it can be used to compute the first row $E[1, j], j = 1, 2, \ldots, M$. For any entry $E[i, j]$, its value can be computed by entries in the previous row $E[i-1, l], i-1 \leq l \leq j-1$. These entries are shaded in Fig. 6.6. Thus, a series of optimal sub-solutions $E[2, j], E[3, j], \ldots, E[M, j]$ in each row of the matrix can be computed subsequently. Finally, these sub-solutions lead to the global optimal solution $\min_{1 \leq i \leq M}\{E[i, M]\}$, which maps all M original nodes onto a new partitioning with minimum energy. Note that the

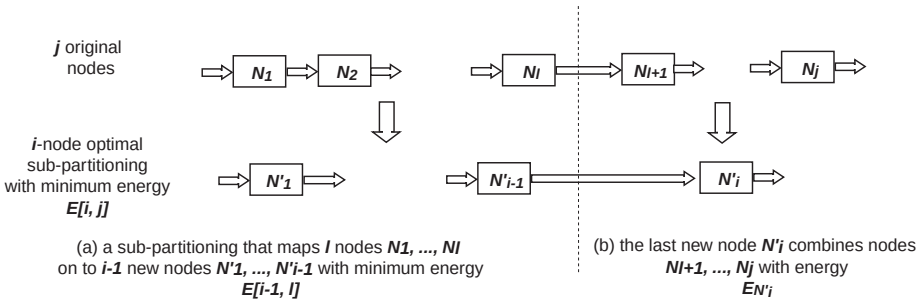


Figure 6.5: The optimal sub-structure of Problem 1.

same algorithm can also solve the optimal partitioning onto a fixed number of nodes. For example, $E[i,M]$ is the optimal energy for mapping M nodes onto an arbitrary i-node new partitioning.

To summarize, the optimal cost function E is defined as follows:

$$E[i,j] = \begin{cases} 0 & \text{for } i = j = 0 \\ \min\limits_{i-1 \le l \le j-1} \left\{ \begin{array}{l} E[i-1,l] \\ +E_{N'_i} \end{array} \right\} & \text{if } \begin{array}{c} U[i-1,l] \\ +\frac{W_{s_j}}{F_{s_j} D} \le 1, \end{array} \\ & \text{for } \begin{array}{c} 1 \le i \le \\ j \le M \end{array} \end{cases} \tag{6.9}$$

To guarantee each optimal sub-solution is schedulable, by Lemma 7, the communication medium must be collision-free, and any node in the new sub-partitioning must not be overloaded. We define a *utilization matrix* $U[i,j]$ indicating the utilization of the communication medium corresponding to the optimal solution of a sub-problem $P[i,j]$, which is guarded by $U[i,j] \le 1$ (Lemma 5). U is initialized to ∞, while setting $U[0,0] = \frac{W_{r_1}}{F_{r_1} D}$ ($= \frac{T_0}{D}$ in(6.7)), indicating the bandwidth used by the first communication instance $RECV_1$. We also define the energy consumption of a node N as E_N that refines (6.5) by Lemma 6. If a node is overloaded, then its energy consumption is ∞ indicating an invalid solution.

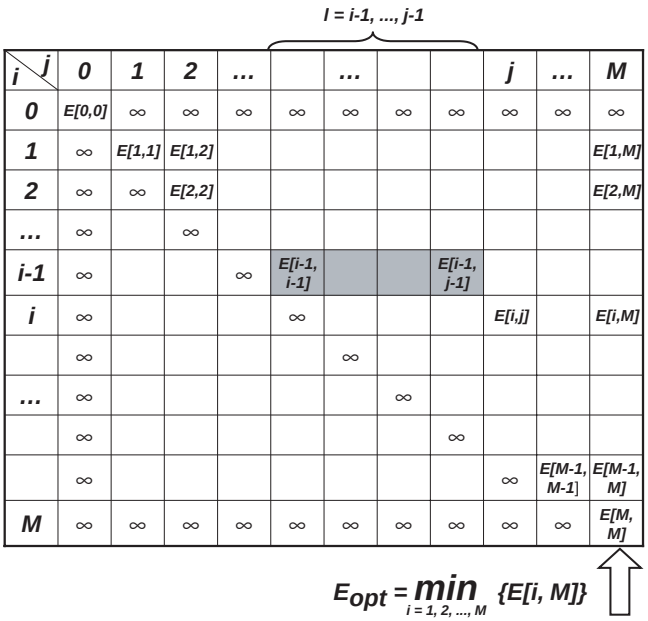


Figure 6.6: The dynamic programming approach to solve Problem 1. Each entry $E[i, j]$ can be computed by the shaded entries in the previous row. The global optimal energy is the minimum value of the last column.

```
partitioning(W_r[1:M], W_s[1:M], W_p[1:M], F_r[1:M], F_s[1:M],
             scale_r, scale_s, scale_p, D, P_ovh)
    for i := 0 to M do
        for j := i to M do
            E[i,j] := U[i,j] := P[i,j] := ∞
    E[0,0] := 0
    U[0,0] := W_r[1]/F_r[1]/D
    for i := 1 to M do
        for j := i to M do
            for l := i-1 to j-1 do
                e := E[i-1,l] + E_{N'_i}
                u := U[i-1,l] + W_s[j]/F_s[j]/D
                if u ≤ 1 and e < E[i,j] then
                    E[i,j] := e
                    U[i,j] := u
                    P[i,j] := l
    E_opt, P_opt := retrieve from matrices E, P
    return E_opt, P_opt
```

Figure 6.7: Optimal partitioning algorithm.

$$U[i,j] = \begin{cases} \frac{W_{r_1}}{F_{r_1} D} & \text{for } i = j = 0 \\ U[i-1,l] + \frac{W_{s_j}}{F_{s_j} D} & \text{for } l \text{ that achieves } \min\{E[i,j]\} \text{ in (6.9)}, \\ & \text{for } 1 \le i \le j \le M \end{cases} \tag{6.10}$$

$$E_N = \begin{cases} scale_r(F_r)T_r + \\ scale_s(F_s)T_s + \\ scale_p(F_p)T_p + \\ P_{ovh}D & \text{if } F_p = \frac{W_p}{D - T_r - T_s} \le F_{max} \\ & (T_r = \frac{W_r}{F_r}, T_s = \frac{W_s}{F_s}) \\ \infty & \text{otherwise} \end{cases} \tag{6.11}$$

Fig. 6.7 shows the optimal partitioning algorithm derived from (6.9) and (6.10). The *partitioning matrix* $P[i,j]$ records the previous optimal sub-solutions for each sub-problem. This information can be used to retrieve the optimal partitioning P_{opt}. The

time complexity of this algorithm is $O(M^3)$ determined by the three-level nested loop.

Problem 2 (Optimal Speed Selection) Given

(a) a fixed partitioning scheme with M pipelined nodes N_i with workload $W_{p_i}, W_{r_i}, W_{s_i}$, $i = 1, 2, \ldots, M$,

(b) a deadline D for all nodes, and

(c) the available choices for communication speed settings $F_{c_k}, k = 1, 2, \ldots, C$,

find all processor speeds F_{p_i} and communication speeds F_{r_i}, F_{s_i} that minimize energy E_{sys}.

We also perform dynamic programming as opposed to exhaustive search in $O(C^{M+1})$ solution space. Since communication speeds decide processor speeds, we only select communication speeds for each node. Given that the sending speed and receiving speed are equal for each communication instance, selecting only sending speed is sufficient. We define a sub-problem $S[i,k]$ that selects communication speeds for the first i nodes, with the last node N_i's sending speed selected to be the k^{th} choice of speed settings, $F_{s_i} = F_{c_k}$. Its optimal sub-solution has minimum energy $E[i,k]$. As illustrated in Fig. 6.8, a sub-problem $S[i,k]$ consists of two parts: (a) another sub-problem $S[i-1,m]$ that selects speed settings for the first $i-1$ nodes with node N_{i-1}'s sending speed $F_{s_{i-1}} = F_{c_m}$, combined with (b) node N_i with receiving speed $F_{r_i} = F_{c_m}$ and sending speed $F_{s_i} = F_{c_k}$. (a) must be an optimal sub-solution with minimum energy $E[i-1,m]$. (b) has only one node N_i that receives data from (a) through speed F_{c_m}; and its sending speed is F_{c_k}. Its energy is denoted as $E_{N_i}(F_r = F_{c_m}, F_s = F_{c_k})$. Therefore, $E[i,k] = E[i-1,m] + E_{N_i}(F_r = F_{c_m}, F_s = F_{c_k})$. In the sub-problem $S[i-1,m]$, F_{c_m} can be any choice among $F_{c_1}, F_{c_2}, \ldots, F_{c_C}$. In order to achieve the minimum energy $E[i,k]$, it must be the minimum value among all possible F_{c_m}. That is, the optimal sub-structure of this problem can be defined as $E[i,k] = \min_{1 \le m \le C}\{E[i-1,m] + E_{N_i}(F_r = F_{c_m}, F_s = F_{c_k})\}$

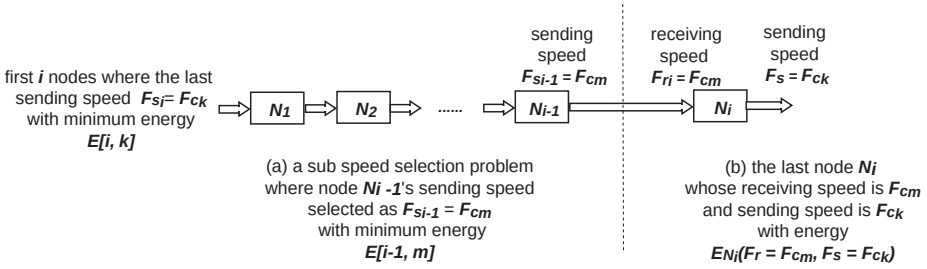


Figure 6.8: The optimal sub-structure of Problem 2.

i \ k	1	2	...	k	...	C
0	E[0,1]	E[0,2]				E[0,C]
1	E[1,1]	E[1,2]				E[1,C]
2	E[2,1]	E[2,2]				E[2,C]
...						
i-1	E[i-1,1]			E[i-1,i-1]		E[i-1,C]
i				E[i,k]		E[i,C]
...						
M	E[M,1]					E[M,C]

$$E_{opt} = \min_{k=1,2,\ldots,C} \{E[M,k]\}$$

Figure 6.9: The dynamic programming approach to solve Problem 2. Each entry $E[i,k]$ can be computed by the shaded row $E[i-1,l]$. The global optimal energy is the minimum value of the last row.

The dynamic programming algorithm is illustrated in Fig. 6.9. Since each $E[i,k]$ can be derived from the previous row $E[i-1,m], m = 1,2,\ldots,C$, the algorithm can compute all rows of matrix E from $E[0,k], E[1,k], \ldots,$ to $E[M,k], k = 1,2,\ldots,C$ sequentially. The global optimal energy is the minimum value in the last row, $\min_{1\leq k\leq C}\{E[M,k]\}$.

The *energy matrix* $E[i,k]$ and *utilization matrix* $U[i,k]$ are defined as follows. $U[i,k] \leq 1$ guarantees that each optimal sub-solution $E[i,k$ is schedulable. Both E and U are initialized to ∞, except $E[0,k] = 0$, $U[0,k]$ is set to the utilization of the first communication instance $RECV_1$ using communication speed F_{c_k}, for $k = 1,2,\ldots,C$.

$$E[i,k] = \begin{cases} 0 & \text{for } \begin{array}{l} i = 0, \\ 1 \le k \le C \end{array} \\ \min\limits_{1 \le m \le C} \left\{ \begin{array}{l} E[i-1,m] + \\ E_{N_i}(F_r = F_{c_m}, \\ F_s = F_{c_k}) \end{array} \right\} & \text{if } \begin{array}{l} U[i-1,m] \\ + \frac{W_{s_i}}{F_{c_k} D} \le 1, \end{array} \\ & \text{for } \begin{array}{l} 1 \le i \le M, \\ 1 \le k \le C \end{array} \end{cases} \tag{6.12}$$

$$U[i,k] = \begin{cases} \frac{W_{r_1}}{F_{c_k} D} & \text{for } \begin{array}{l} i = 0, \\ 1 \le k \le C \end{array} \\ U[i-1,m] + \frac{W_{s_i}}{F_{c_k} D} & \begin{array}{l} \text{for } m \text{ that achieves} \\ \min\{E[i,k]\} \text{ in (6.12)}, \end{array} \\ & \text{for } \begin{array}{l} 1 \le i \le M, \\ 1 \le k \le C \end{array} \end{cases} \tag{6.13}$$

The algorithm is shown in Fig. 6.10. The *speed matrix S* records the previous optimal sub-solutions. The optimal speed setting S_{opt} will be retrieved from S. The time complexity of this algorithm is $O(MC^2)$. Note that the algorithm can be modified trivially to if the first communication speed F_{r_1} and the last communication speed F_{s_M} are fixed. This refers to the situation where the pipelined multi-processor has a fixed communication speed setting to other components while its "internal" communication speeds can be selected to optimal.

Problem 3 (Optimal Partitioning and Speed Selection) Given

(a) M pipelined nodes N_i with workload $W_{p_i}, W_{r_i}, W_{s_i}$, $i = 1, 2, \ldots, M$,

(b) a deadline D for all nodes, and

```
speedselection(W_r[1:M], W_s[1:M], W_p[1:M], F_c[1:C],
               scale_r, scale_s, scale_p, D, P_ovh)
    for i := 1 to M do
        for k := 1 to C do
            E[i,k] := U[i,k] := S[i,k] := ∞
    for k := 1 to C do
        E[0,k] := 0
        U[0,k] := W_r[1]/F_c[k]/D
    for i := 1 to M do
        for k := i to C do
            for m := i to C do
                e := E[i-1,m] + E_{N_i}(F_r = F_c[m], F_s = F_c[k])
                u := U[i-1,m] + W_s[i]/F_c[k]/D
                if u ≤ 1 and e < E[i,m] then
                    E[i,k] := e
                    U[i,k] := u
                    S[i,k] := m
    E_opt, S_opt := retrieve from matrices E, S
    return E_opt, S_opt
```

Figure 6.10: Optimal speed selection algorithm.

(c) the available choices for communication speed settings $F_{c_k}, k = 1, 2, \ldots, C$,

find a partitioning scheme and corresponding communication speed settings that minimize energy E_{sys}.

Due to the inter-dependency between speed settings and partitioning schemes, the optimal solution cannot be achieved by solving two previous problems individually. Exhaustively enumerating over one dimension and dynamic programming over the other is quite expensive with the time complexity as either $O(2^{M-1}MC^2)$ or $O(C^{M+1}M^3)$. We proposed a *multi-dimensional dynamic programming* algorithm given the fact that the two previous problems are all characterized by optimal sub-structures. Based on the dynamic programming approaches in previous problems, we define a sub-problem $PS[i, j, k]$ that maps j original nodes $N_1, N_2, \ldots, N_j$ onto an i-node new sub-partitioning $N'_1, N'_2, \ldots, N'_i$, with the last node N'_i's sending speed $F'_{s_i} = F_{c_k}$. The optimal sub-solution has minimum energy $E[i, j, k]$.

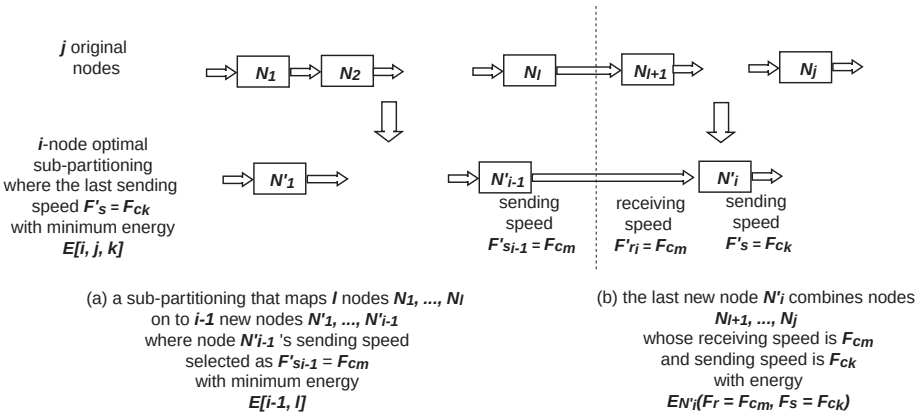


Figure 6.11: The optimal sub-structure of Problem 3.

Similar to the previous problems, a sub-problem $PS[i, j, k]$ can be decomposed with an optimal sub-structure, shown in Fig. 6.11. (a) is a previous sub-problem $PS[i-1, l, m]$, which maps the first l original nodes $N_1, N_2, \ldots, N_l$ onto $i-1$ new nodes with node N'_{i-1}'s sending speed selected as F_{c_m}. (b) is the new node N'_i that combines original nodes $N_{l+1}, \ldots, N_j$ with receiving speed F_{c_m} and sending speed F_{c_k}. (a) must be an optimal sub-solution with the minimum energy $E[i-1, l, m]$. Note that (b) has only one node N'_i, and its energy is denoted as $E_{N'_i}(F_r = F_{c_m}, F_s = F_{c_k})$. For sub-solution $E[i-1, l, m]$, l can be any value in range $i-1 \leq l \leq j-1$ and F_{c_m} is one of C speed choices $F_{c_1}, F_{c_2}, \ldots, F_{c_C}$. $E[i, j, k]$ must be derived from all possible pairs of (l, m) to achieve the minimum value. Therefore, $E[i, j, k] = \min_{i-1 \leq l \leq j-1, 1 \leq m \leq C}\{E[i-1, l, m] + E_{N'_i}(F_r = F_{c_m}, F_s = F_{c_k})\}$.

The algorithm is illustrated in Fig. 6.12. The three-dimensional matrix $E[i, j, k]$ is represented by a series of two-dimensional sub-matrix indexed by $i = 0, 1, \ldots, C$. Any $E[i, j, k]$ can be computed from entries in a sub-matrix $E[i-1, l, m], i-1 \leq l \leq j-1, 1 \leq m \leq C$. The algorithm constructs all optimal sub-solutions from $E[0, j, k], E[1, j, k], \ldots$ to $E[M, j, k], 1 \leq j \leq M, 1 \leq k \leq C$. The global minimum energy is $\min_{1 \leq i \leq M, 1 \leq k \leq C}\{E[i, M, k]\}$. It refers to the minimum value of the last rows in all sub-matrices.

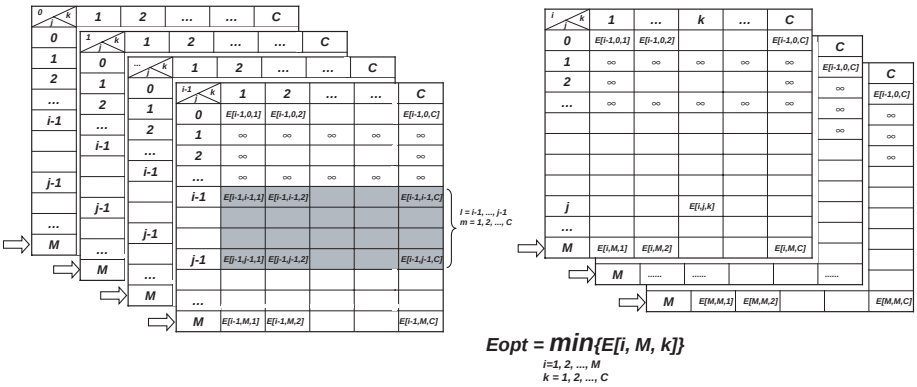


Figure 6.12: The multi-dimensional dynamic programming approach to solve Problem 3. Each entry $E[i,j,k]$ can be computed by the shaded entries in the previous sub-matrix. The global optimal energy is the minimum value in the last row of all sub-matrices.

The *energy matrix* $E[i,j,k]$ and the *utilization matrix* $U[i,j,k]$ is defined as follows.

$$E[i,j,k] = \begin{cases} 0 & \text{for } i=j=0,\ 1\le k\le C \\ \min\limits_{\substack{i-1\le l\\ \le j-1,\\ 1\le m\le C}} \left\{ \begin{array}{l} E[i-1,l,m]+ \\ E_{N_i'}(F_r=F_{c_m},\\ F_s=F_{c_k}) \end{array} \right\} & \text{if } \begin{array}{l} U[i-1,l,m] \\ +\frac{W_{s_j}}{F_{c_k}D}\le 1, \\ 1\le i\le \\ \text{for } j\le M, \\ 1\le k\le C \end{array} \end{cases} \tag{6.14}$$

$$U[i,j,k] = \begin{cases} \frac{W_{r_1}}{F_{c_k} D} & \text{for } i = j = 0, \quad 1 \le k \le C \\ U[i-1,l,m] + \frac{W_{s_j}}{F_{c_k} D} & \text{for } (l,m) \text{ that achieve } \min\{E[i,j,k]\} \text{ in (6.14)}, \quad 1 \le i \le j \le M, \quad 1 \le k \le C \end{cases} \tag{6.15}$$

The algorithm is shown in Fig. 6.13. It combines two previous algorithms by two-dimensional dynamic programming. The time complexity of the algorithm is $O(M^3C^2)$. It also applies to situations where the new partitioning has a fixed number of nodes, or the pipeline has a fixed communication interface to other components while only internal communication speed can be selected.

6.7 Analytical Results

To evaluate our energy optimization technique, we experimented with mapping the ATR algorithm [61] (Fig. 6.3) onto two fixed partitioning schemes: (a) a single-node that combines all blocks, and (b) a five-node pipeline that maps each block onto an individual node. (a) and (b) are two extremes representing serial vs. parallel schemes. For both (a) and (b) we apply optimal speed selection. We also find the optimal partitioning with speed selection as (c) and compare with (a) and (b) under three types of performance requirements: (1) high performance, $D = 10ms$, (2) moderate performance, $D = 15ms$, and (3) low performance, $D = 20ms$.

Each node consists of an XScale processor and an LXT-1000 Ethernet interface from Intel. The $Scale_p$ and $Scale_s$ (same as $Scale_r$) functions, which indicate the power vs. performance characteristics of a node, are extracted from their data sheets [2, 3]

```
partitioning-speedselection(W_r[1:M], W_s[1:M], W_p[1:M],
                F_c[1:C], scale_r, scale_s, scale_p, D, P_ovh)
    for i := 0 to M do
        for j := i to M do
            for k := 1 to C do
                E[i,j,k] := U[i,j,k] := P[i,j,k] := S[i,j,k] := ∞
    for k := 1 to C do
        E[0,0,k] := 0
        U[0,0,k] := W_r[1]/F_c[k]/D
    for i := 1 to M do
        for j := i to M do
            for k := 1 to C do
                for l := i-1 to j-1 do
                    for m := 1 to C do
                        e := E[i-1,l,m] + E_node(merge(N_{l+1},...,N_j),
                                with F_r = F_c[m], F_s = F_c[k])
                        u := U[i-1,l,m] + W_s[j]/F_c[k]/D
                        if u ≤ 1 and e < E[i,j,k] then
                            E[i,j,k] := e
                            U[i,j,k] := u
                            P[i,j,k] := l
                            S[i,j,k] := m
    E_opt, P_opt, S_opt := retrieve from matrices E, P, S
    return E_opt, P_opt, S_opt
```

Figure 6.13: Combined partitioning with speed selection.

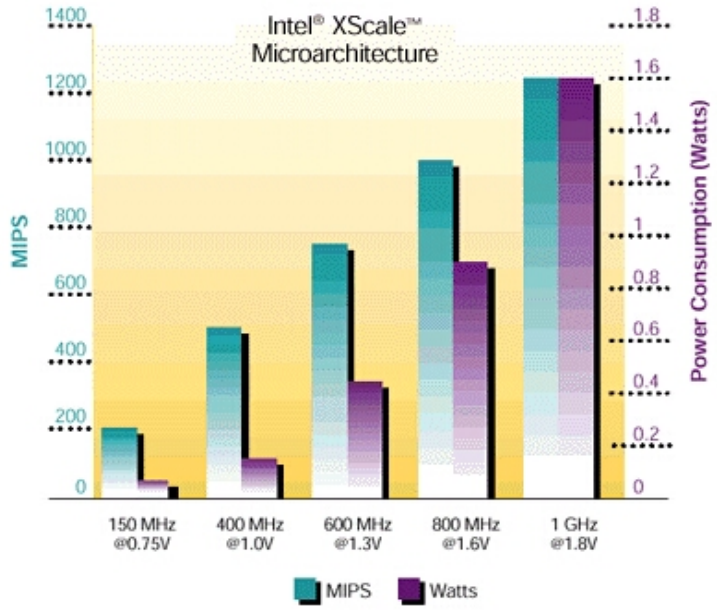


Figure 6.14: Power vs. performance of the XScale processor.

Mode	*Power consumption*
10M bps	800 mW
100M bps	1.5W
1000M bps	6W

Figure 6.15: Power modes of the Ethernet interface.

and are shown in Fig. 6.14 and 6.15. Besides the power draw from the CPU and communication interfaces, we assume each node has a constant power draw $P_{ovh} = 100mW$.

The results are presented in Fig. 6.16. In all cases, 1000Mbps is always the optimal speed setting for communication. The low-power, 10Mbps communication speed results in the highest energy. This is because it leaves so little time for computation such that the processors must run faster with more energy to meet the deadline, and it has the highest energy-per-bit rating. The low-speed communication also tends to violate the

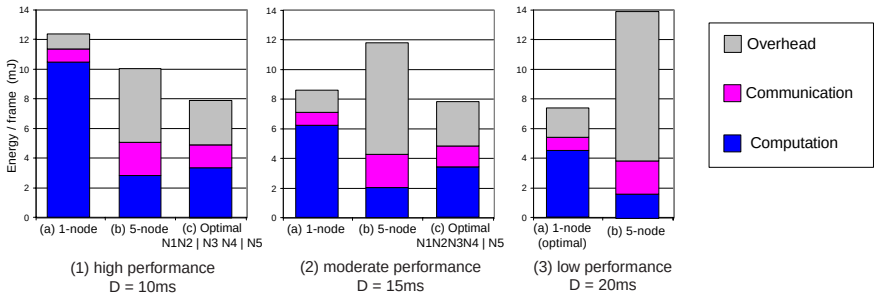


Figure 6.16: Analytical results.

schedulability conditions (Lemma 7). Given properties of this particular Ethernet interface, 1000Mbps communication will always lead to the lowest energy consumption since it requires the least amount of energy per bit and leaves the maximum amount of time budget for reducing CPU energy. However, in cases where the energy-per-bit rating does not decrease monotonically with the communication speed, the optimal speed setting may involve some combinations of low-speed and high-speed settings between different nodes. For example, the node N_i may communicate with N_{i-1} at 1000Mbps and with N_{i+1} at 100Mbps.

Fig. 6.16(1) shows the energy consumption per image frame in three partitioning schemes. With a tight performance constraint, the single-node (a) is heavily loaded with computation. Therefore it is desirable to reduce CPU energy by pipelining. As a result, the five-node pipeline (b) is more energy-efficient at the cost of additional communication and overhead. However, the optimal partitioning is (c) with three nodes: $[N1,N2],[N3,N4],[N5]$. It consumes more CPU energy than (b), but overall it is optimal with less energy on communication and overhead.

In case of the moderate performance constraint (Fig. 6.16(2)), (a) is still dominated by computation but it is not heavily loaded due to the relaxed deadline. The reduction of CPU energy by (b) cannot compensate for the added overhead of new nodes and communication. Therefore (a) is better than (b) and pipelining seems inefficient. However, the optimal partitioning (c) is still a pipelined solution. It combines $N1,N2,N3,N4$ into

one node and maps $N5$ to another node. (c) achieves minimum energy by appropriately balancing computation, communication with pipelining overhead.

In cases where the performance is not critical, pipelining is not efficient and the serial solution (a) is optimal. Fig. 6.16(3) shows that the computation load on (a) is very light. Introducing additional nodes will only save marginal CPU energy that will be offset by extra communication and overhead.

6.8 Chapter Summary

We present a combined partitioning and speed selection technique for the energy optimization of embedded multiprocessor-on-chip architectures with high-speed on-chip networks. As communication power approaches or surpasses that of processor power, communication must be treated as a primary concern in system-level energy optimization. We exploit the multi-speed feature of modern high-speed communication interfaces as an effective way to complement and enhance today's CPU-centric power optimization approaches. In such systems, communication and computation compete over opportunities for operating at the most energy-efficient points. It is critical to not only balance the load among processors by functional partitioning, but also to balance the speeds between communication and computation on each node and across the whole system.

Our multi-dimensional dynamic programming formulation is exact and is of polynomial time complexity. It produces energy-optimal solutions as defined by a partitioning scheme and by the speed selections for all computation and communication tasks. We expect this technique to be applicable to a large class of data dominated systems-on-chip that can be structured in a pipelined organization.

Part IV

Mode Selection

Chapter 7

Power Mode Selection

Among the techniques for system-level power management, it is not currently possible to guarantee timing constraints and have a comprehensive system model supporting multiple components at the same time. We propose a new method for modeling and selecting the power modes for the optimal system-power management of embedded systems under timing and power constraints. First, we not only model the modes and the transitions overhead at the component level, but we also capture the application-imposed relationships among the components by introducing a *mode dependency graph* at the system level. Second, we propose a mode selection technique, which determines when and how to change mode in these components such that the whole system can meet all power and timing constraints. Our constraint-driven approach is a critical feature for exploring power/performance tradeoffs in *power-aware* embedded systems. We demonstrate the application of our techniques to a low-power sensor and an autonomous rover example.

7.1 Introduction

Recent trends in mobile and autonomous embedded systems are giving rise to a new class of *power-aware* systems. Unlike low-power systems, whose goal is to minimize power usage, power-aware systems are more general in that they must make the best use of the available power by adapting their behavior to the constraints imposed by the environment, user requests, or their power sources. Power-aware systems must use components that are capable of multiple modes of operation. Many of these components offer modes for power management, while other components allow the user to control the voltage or frequency as other forms of power modes. The selection of mode is thus the primary means of controlling power usage, and it is often done in conjunction with scheduling.

New off-the-shelf components are offering increasingly sophisticated modes for power management. However, the system-level power manager has only limited control over the modes. Some modes can be set by writing commands to a control register of a device. However, the power manager may not be able to arbitrarily select the modes it wishes at all times. It may be forced to wait or request a change through a sequence of intermediate modes. Even if a desired mode is available, changing mode can incur nontrivial overhead both in terms of time and power. The overhead translates into penalty in performance or power, and it can cause a system to miss an important deadline.

Another key issue for power management is that mode selection cannot be done in isolation. The choice of mode in one component must be coordinated with that in other components, or else the whole system may not function correctly. For example, if the mode selection involves a particular encoding scheme, then the rest of the system that depends on the data representation must also change mode in order to handle the encoding correctly.

It can be difficult for designer to track details with modes. The problem is further

exacerbated by the fact that the number of components and the available modes are increasing rapidly. Today's methodologies either limit the complexity by using only a small subset of the available modes (e.g., on, sleep, off), or they are unable to guarantee timing or power constraints.

Power management of embedded systems must consider all components in the system. Significant power reduction in one components may not translate into desirable power reduction for the whole system. In mission critical applications, peripheral devices including mechanical and thermal devices can actually dominate power consumption and must be an integral part of power management.

We believe that a new methodology for mode modeling and selection is sorely needed in order to effectively manage the power of the next generation embedded systems. We first introduce a new *mode dependency graph* for modeling the *enabling* relationships among modes within a component and between components in a system. Second, we present a new mode selection algorithm that produces a mode schedule that satisfies timing and power constraints on multiple processors and devices. It takes advantage of the mode dependency graph in effectively pruning the search space, making it practical to incorporate into an on-line power manager. The advantage with our constraint-driven approach is that it is not hardwired to a specific objective such as power minimization. This is a crucial feature for power-aware embedded systems, for which the ability to make power/performance tradeoffs is more important than just power reduction.

This chapter is organized as follows. Section 7.2 reviews related work. Section 7.3 presents the mode dependency graph, while Section 7.4 describes a mode selection algorithm that takes advantage of mode dependency modeling. We discuss the experimental results in Section 7.5.

7.2 Related Work

Many low-power techniques have been developed at all levels. For system-level designs, since the components are largely off-the-shelf or already designed, the applicable techniques include dynamic voltage scaling (DVS) and dynamic power management (DPM).

7.2.1 Dynamic Voltage Scaling (DVS)

Developed for variable-voltage processors, DVS can achieve significant energy saving while still enabling the processor to continue making progress [74, 25]. Although DVS means running slower, they typically slow down just enough without violating timing constraints, and many are based on real-time task scheduling cores [25, 59, 60, 54].

It has been shown that maximal energy saving is achieved by running the processor at the slowest possible constant speed, rather than running tasks at full processor speed and changing the processor to a lower power mode when idle [14]. Hong et al [25] proposed a heuristic for scheduling real-time tasks on a variable voltage processor. Shin [59] exploited both execution time variation and idle time intervals for fix-priority tasks. Shin's algorithm in [60] determines the lowest maximum processor speed for each job to achieve power reduction. Quan and Hu [54] further greedily determine the lowest voltage for a set of tasks to achieve more energy savings.

What these DVS techniques have in common is that they are greedy and assume a single processor. A power-aware embedded system, however, consists of multiple resources, which may be one or more processors and peripheral devices. Unfortunately, greedy DVS techniques are not generalizable to multiple resources under power constraints, as shown in the following example.

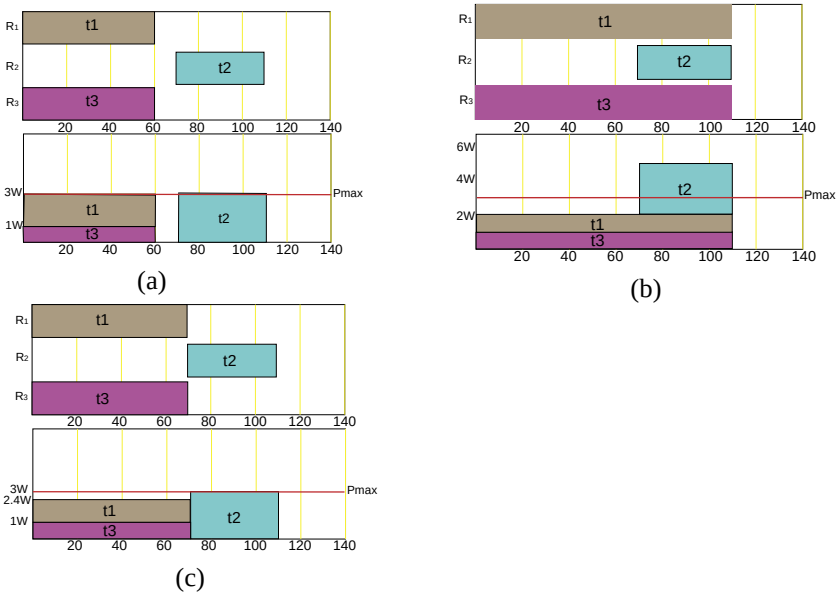


Figure 7.1: An application scenario that has resource dependency.

Example: (DVS fails in multi-resource)

Fig. 7.1(a) shows a Gantt chart (top) and the power profile (bottom) for a system with three resources: R_1 is capable of voltage scaling, while R_2 and R_3 are not. The task t_1 on R_1 has a deadline at 110. The system has a max power constraint of $3W$. Furthermore, the behavior of the application dictates that R_1 and R_3 be *co-active*. Co-activation means the execution of one task requires the power consumption of other dependent services or tasks. A simple example is that when the CPU is running, it imposes a co-activation dependency on the memory, but co-activation can be much more general between sets of tasks.

Fig. 7.1(b) shows the schedule and the power profile obtained by greedily slowing down R_1. Even though all timing constraints are satisfied, it violates power constraints and it is not minimum energy. When it is stretched out, t_1 overlaps t_2 during time 70-110, and their total power exceeds the max power constraint. It is not minimum energy due to the co-activation dependency between R_1 and R_3: the energy saving by R_1 due

Schedule	Timing violation	Power violation	Energy cost
Fig. 7.1(a)	No	No	300
Fig. 7.1(b)	No	Yes	320
Fig. 7.1(c)	No	No	288

Figure 7.2: Comparison of three schedules.

to voltage scaling is more than offset by R_3, whose execution is prolonged by R_1.

The optimal schedule and power profile are shown in Fig. 7.1(c). Resource R_1 is slowed down without overlapping t_2 on R_2. No max power is violated. Although t_3 is stretched with t_1 and therefore consumes more energy than in Fig. 7.1(a), t_1 saves even more energy due to voltage scaling of resource R_1. As a result, the system achieves minimal energy while satisfying all constraints. Fig. 7.2 summarizes the energy costs.

Another problem not highlighted with this example is that mode changes may incur nontrivial power or timing overhead. If so, overhead must be considered in determining the feasibility of the mode schedule.

Luo and Jha [45] presents scheduling for multiple processing elements by reordering tasks and applying voltage scaling in this post-processing step after scheduling. Our approach is similar in that it can also be a post processing step, and handles precedence and timing constraints, but we treat power as a hard constraint. Furthermore, we handle co-activation and other mode-dependency relationships.

7.2.2 Dynamic power management (DPM)

Previous work on DPM mainly aimed to achieve power reduction by predicting the system idle time or event distribution and shutting down resources when idle. The simplest power management policy is time-out based on a fixed or predicted amount of time before the system's shutdown or powerup [30]. Stochastic model [12] is used to address the uncertainty in system behaviors. DPM techniques can be effective for minimizing energy and time penalties on average, but they have several limitations. First, most treat either power or timing as an *objective* or penalty, rather than a *con-*

straint. In real systems, the max power is a real, hard constraint, whose violation can lead to malfunction. Second, they have not considered inter-component dependency in a system, with the exception of Qiu, Qu and Pedram in [53], which models multiple service providers and their GSPN model can capture some dependencies among resources. However, their model is mainly for the request/dispatch behavior of servers rather than dependency among the servers themselves.

Our new approach, mode selection, combines the advantages of existing approaches. It is entirely constraint driven, enabling us to make power/performance tradeoffs without hardwiring any specific goal or policy in the algorithm.

7.3 Modeling Resource Dependency

Selecting (or not selecting) a mode of a resource may impact the modes that other resources are allowed to select. The impact may be co-activation, exclusion, enabling, and many other possible types of dependency. These dependencies may be extracted from application level specifications or policies for safety, security, fault-tolerant, power-saving. In any case, a *legal* mode combination of the resources is one that respects all of these dependencies, and a *feasible* mode combination is one that is legal and satisfies all the constraints (namely timing and power). We use a data structure called the mode dependency graph (MDG) that enables efficient generation of legal mode combinations in an order that facilitates the search for feasible combinations that are also low cost.

7.3.1 Definitions

Definition 15 (Resource $\gamma \in \Gamma$) A resource γ is defined as a graph $R_\gamma(M_\gamma, H_\gamma)$, where M_γ is a set of vertices, and $H_\gamma \subseteq M_\gamma \times M_\gamma$ is a set of edges. A vertex $m \in M_\gamma$ is a power mode of resource γ. An edge $(m,n) \in H_\gamma$ represents a mode change from mode m to mode n. We define the timing and energy function for a mode change as: F :

$M_\gamma \times M_\gamma \rightarrow T \times E$, where M_γ is the set of modes of resource γ, and T, E are time and energy, respectively. The average power can be obtained from energy and time information.

Definition 16 (Power and delay functions) Power consumption of a resource γ is represented as a function, π, mapping from power mode to a power number. Formally, $\pi : M_\gamma \rightarrow \mathbb{R}^+$. Delay of a mode transition is defined as a function, δ, mapping from start mode and end mode of a transition to a delay number. Formally, $\delta : M_\gamma \times M_\gamma \rightarrow \mathbb{R}^+$.

Definition 17 (Mode combination $\lambda \in \Lambda$) Given N resources $(\gamma_1, \gamma_2, \ldots, \gamma_N)$, a mode combination is $\lambda \in M_{\gamma_1} \times M_{\gamma_2} \times, \ldots, M_{\gamma_N}$.

7.3.2 Mode Dependency Graph

A mode dependency graph (MDG) $G(M,D)$ characterizes the inter-resource dependency relationships, where $M = \bigcup_{\gamma \in \Gamma} M_\gamma$ is a set of vertices representing power modes, and D is a set of edges standing for dependencies. A vertex is represented by a circle with a label in the format of "$\gamma.m$," where $\gamma \in \Gamma$ is a resource and $m \in M$ is a mode of the resource. If two vertices have the same labels, we considered them identical.

The *value* of a vertex $v \in M$ is defined as:

$$|v| = \begin{cases} True & \text{if } \gamma \text{ is in mode } m, \\ False & \text{if } \gamma \text{ is in other mode,} \\ Undetermined & \text{if } \gamma \text{ has not been selected a mode.} \end{cases} \tag{7.1}$$

An edge in the MDG represents dependency between two modes. Suppose an edge $(u,v) \in E$, $u = \gamma_1.m_1$, $v = \gamma_2.m_2$. The two modes m_1 and m_2 satisfy the mode dependency graph if:

$$\begin{aligned} &|u| \text{ is } True \text{ only if } |v| \text{ is } True \ (\dashrightarrow), \\ &|v| \text{ is } False \text{ implies } |u| \text{ is } False \ (\Leftarrow). \end{aligned} \tag{7.2}$$

u	v	Violation
True	*False*	YES
True	*True*	NO
False	*True*	NO
False	*False*	NO

Figure 7.3: A table for violation checking.

```
Check_MDG(mode dependency graph G, resource γ, mode m):
1   V ← find all vertices v ∈ V that contain resource γ
2   for each v ∈ V {
3       find vertex u such that u points to v, if any
4       if violation checking for (u, v) according to Fig. 7.3 is YES {
5           return False
6       }
7   }
8   return True
```

Figure 7.4: Check satisfaction of an MDG.

In other words, if $|u|$ is *True*, $|v|$ *may* be *True*; but if $|v|$ is *False*, $|u|$ *must* be *False*. For example, we represent the dependency between a CPU and a memory chip such that the memory is on only if the CPU is in active mode. If the CPU is not in active mode, then the memory must not be on. If both of the above conditions are met, we say that the CPU and the memory satisfy the mode dependency. Otherwise, they violate the mode dependency. Fig. 7.3 summarizes the conditions that (do not) violate the mode dependency.

To expand the capability of mode dependency graph, we introduce the logic operators as another kind of vertices. An operator vertex is represented by a square with an operator label in it. For the operator vertex with multiple outgoing edges, the $\dashrightarrow$ direction combines disjunctively, and the $\Leftarrow$ direction combines conjunctively. For example, a vertex u, whose value $|u|$ is *True*, points two vertices v_1 and v_2. If either v_1 or v_2, or both, are *True*, then they satisfy mode dependency. When v_1 and v_2 are both false, they violate mode dependency. The value of an operator vertex can be obtained

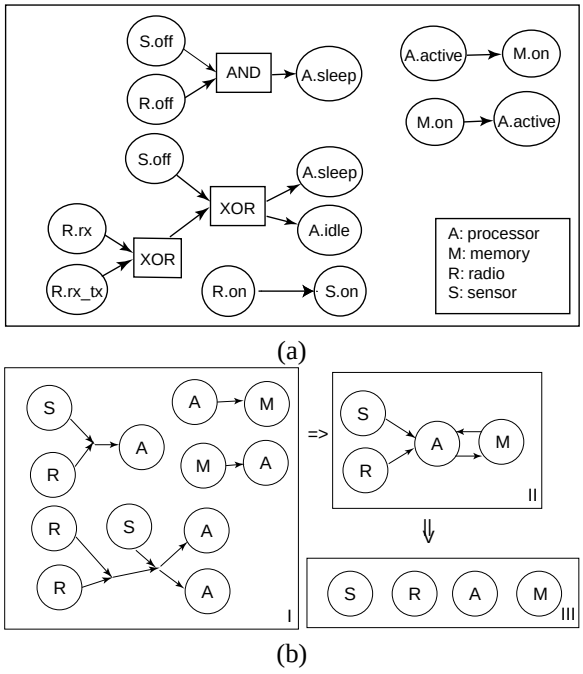


Figure 7.5: (a) An MDG example: microsensor. (b) Reduce the MDG to a resource list.

by evaluating the logic function it represents. We define the operators *AND*, *OR* and *XOR*. The functions of the operators follow the normal boolean functions in the same names except when any input is "undetermined," the output is "undetermined." Given an MDG, a resource γ and one of its mode *m*, we can use the routine in Fig. 7.4 to check whether mode *m* satisfies the MDG.

7.3.3 Generating Mode Combinations

This section shows how to efficiently generate legal mode combinations using the MDG.

We transform and reduce an MDG to a resource list. The purpose is to sequence the resources so that the modes of a resource do not depend on those of the succeeding resources. From the MDG, we shrink each operator vertex to a point, and remove mode

name in each mode vertex. We then remove the redundant vertices and edges, break the cycle by removing one edge in the cycle, and apply topological sort to obtain a resource list.

If the MDG is acyclic, then legal mode combinations can be generated by a special version of topological traversal. Starting from the first resource of the list, we check modes of each resource against the MDG and identify the legal modes. We keep them and select one for the current resource γ, and move to the next resource. We are able to determine a mode of γ because upon checking the resource, all the modes of its dependent resources have been already determined since they are all located before γ. We progressively generate a mode combination as we check legality of modes and select one at each resource. As we reach the end of the list, we obtain a legal mode combination. We enumerate the rest of legal modes at the end resource, backtrack to previous resources, and enumerate their legal modes to generate other legal mode combinations.

Note that there may be cycles in an MDG, which implies that in the resource list obtained above, modes of a resource may depend not only on preceding resources, but also on succeeding resources. We call such resources *dirty-resource*. In this scenario, we keep track of which resources the current resource γ are dependent on. When the modes of all dependent resources are determined, we evaluate a mode of γ to determine whether the mode satisfies the MDG. Fig. 7.6 shows the detailed algorithm, which is the general case for both acyclic and cyclic MDGs.

7.3.4 Example: Microsensor

A microsensor system is a node in a distributed microsensor network [63]. It consists of a sensor, a processor, memory chips, radio frequency module and other auxiliary parts. The microsensor obtains information from environment and sends processed data to a base station. The sensor and the memory each has two modes, on and off.

```
MODEGEN_FROM_CYCLIC_MDG(mode dependency graph G):
1   reset the list of mode combinations Λ ← ∅
2   reset a mode combination λ ← ∅
3   transform G into a resource list L[0...N−1
4   mark dirty-resources in L
5   p ← 0
6   while p ≥ 0 {
7       while 0 ≤ p < N {
8           if L[p] is not a dirty-resource {
9               if found an unmarked mode m for task L[p] {
10                  if check_MDG(G, γ, m) = True {
11                      if L[p].cached is not empty {
12                          if all cached resources satisfy G {
13                              λ[p] ← m, p ← p+1 }
14                      } else {λ[p] ← m, p ← p+1 }
15                  }
16                  mark the mode m
17              } else {unmark all modes of L[p], p ← p−1 }
18          } else {
19              locate the last resource L[q] that L[p] is dependent on
20              L[q].cached ← L[p]
21          }
22          p ← p+1
23      }
24      if p = length(L) { push λ into Λ}
25      p ← p−1
26      if found an unmarked mode m for the resource L[p] {
27          unmark all modes for current resource
28          p ← p−1
29      } else {
30          if check_MDG(m, G) =True {
31              λ[p] ← m, push λ into Λ, p ← p+1 }
32      }
33  }
34  return Λ
```

Figure 7.6: Generate mode combinations for cyclic MDG.

The processor has three modes, active, idle and sleep. The radio has three modes, transmit-and-receive (tx_rx), receive-only (rx), and off. There are a total of 36 mode combinations for these components.

The behavior and dependencies of the devices in this system can be derived from high-level power management policies: the sensor and the radio may be both off only if the processor is in sleep mode; either of them may be on only if the processor is in sleep mode or idle mode; both the sensor and the radio may be on only if the processor is in active mode; the memory is on if and only if the processor is active. Fig. 7.5(a) shows the MDG of the microsensor.

Using the MDG, our algorithm automatically generate eight mode combinations that satisfy the given MDG (see Fig. 7.7). Suppose we want the microsensor to work in a proactive way: when it is off, the system can only be waken up by the sensor when it senses information from environment. The radio cannot wake up the system, for example, by receiving a remote command. We add another item "the radio may be on only if the sensor is on" (in dashed box in Fig. 7.5(a)) to the MDG in Fig. 7.5(a). Then we run our algorithm on the new MDG and obtain five mode combinations (without * in Fig. 7.7). This result exactly matches the mode combinations in manually designed results [63].

Through this simple example, we show our algorithm is able to systematically generate legal mode combinations, and by editing the mode dependency graph, we can obtain mode combinations without manually going through all possible mode combinations.

7.4 Mode Selection

Mode selection works as a post-processing stage after scheduling. It validates and improves the schedule with more architectural knowledge than the scheduler. Our approach is a constraint-driven search algorithm that considers resource/task dependency

mode	S	R	A	M
M1	on	tx_rx	active	on
M2	on	rx	idle	off
M3	on	rx	sleep	off
M4	on	off	sleep	off
*M5	off	tx_tx	active	on
*M6	off	rx	idle	off
*M7	off	rx	sleep	off
M8	off	off	sleep	off

Figure 7.7: Mode combinations of microsensor.

and mode change overhead, and tries to find a mode schedule that satisfies system timing and power constraints.

7.4.1 Problem Statement

The input to the problem consists of a set of tasks X, a schedule σ, a mode dependency graph G, power constraints P_{max} and P_{min}, and timing constraints represented by constraint graph G_c [44]. The output is a mode schedule σ' that meets system power and timing constraints by means of legal mode combinations.

Definition 18 (Task $x \in X$) A *task* x is defined by a tuple (τ_x, ω_x), where τ_x is a task identifier, and $\omega_x \in \Omega$ is the workload of the task. In the context of this chapter, we assume each task x has already been mapped to a resource γ. The operation delay d_x and power profile $P_x(t)$ of a task x depend on the workload ω_x and the selected modes m of resource γ.

Depending on the nature of the resource, workload ω_x can be the number of cycles for a processor, the number of atomic actions for a device, e.g., the number of steps for a step motor, or simply the time to perform a task.

Definition 19 (Schedule σ) A *schedule* σ maps each task to its start time. An *idle interval* with respect to a schedule σ and a resource γ is a time interval during which

no task is scheduled to run on γ. Note that during an idle interval, the resource can still consume nonzero power, depending on the mode.

Definition 20 (Mode schedule σ') A *mode schedule* σ' maps each task $x \in X'$ (which is mapped to resource γ) to the task's start time and a mode $m \in M_\gamma$, where $X' = X \cup X_o$. X_o is a set of *overhead tasks*, which are inserted whenever there is a mode change on a given resource.

A mode schedule σ' is *feasible* if all mode combinations are legal (Section 7.3) and all timing and power constraints are satisfied at all times:

$$P_{min} \leq \sum_{\gamma \in \Gamma} P_\gamma(t) \leq P_{max} \qquad 0 \leq t \leq t_{end} \tag{7.3}$$

$$T_{min}(u,v) \leq \sigma'(v) - \sigma'(u) \leq T_{max}(u,v) \quad \forall u,v \in \text{task set } X \tag{7.4}$$

where t_{end} is the overall schedule length, and P_{min} and P_{max} are the minimum and maximum power constraints, respectively. The reason for a minimum power constraint has been discussed elsewhere [44]. It can be used for not only power/performance tradeoffs but also for jitter control.

7.4.2 Algorithm

Our Mode Selection algorithm contains a loop with two steps. First we find modes for tasks that satisfy task dependency and timing constraints. Second we determine modes for the idle intervals on each resource. Note that after the first step, the operation delay for certain tasks may be changed due to certain mode selected (i.e., modes of different clock rate due to voltage scaling) or task dependency. An advantage of selecting task modes and idle interval modes separately is that we can apply different kinds of system constraints, which help prune out illegal mode combinations efficiently. We reorder the modes for each resource by their power consumption in increasing order and search

```
MODE_SELECTION(σ, G, P_max, P_min, G_c):
0   /* input : schedule σ */
1   /*         power constraints P_max and P_min */
2   /*         timing constraint graph G_c */
3   /*         mode dependency graph G */
4   /* output: a feasible mode schedule σ' */
5   MODEGEN_FROM_CYCLIC_MDG(G)
6   for each λ in Λ {
7       map λ to tasks T in σ, get σ_1
8       if check_timing(σ_1, G_c) = True {
9           decompose σ_1 into time intervals S
10          for each s ∈ S {
11              select modes for idle intervals
12              while P_max and P_min not satisfied {
13                  select other modes for idle intervals
14              }
15          } /* So far we obtain a mode schedule σ' */
16          add mode change overhead as new tasks into σ_1, get σ_2
17          if check_timing(σ_2, G_c) = True and
18              check_power(σ_2, P_max, P_min) = True {
19                  return λ∪ { modes selected for idle intervals }
20          }
21      }
22  }
```

Figure 7.8: Top level Mode Selection algorithm.

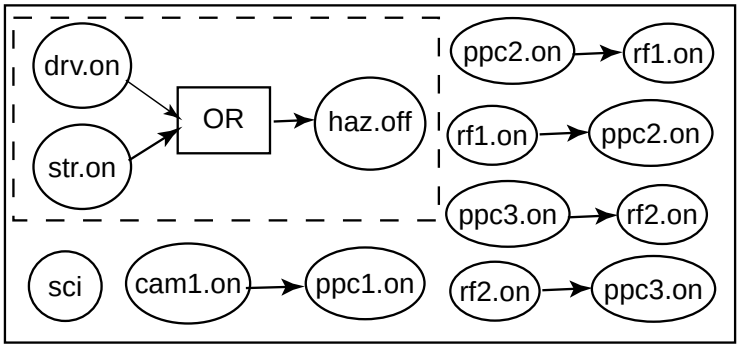


Figure 7.9: The MDG for the Microrover.

from the smallest one. By doing so we both speed up our search process and find solutions very close to the energy-optimal solution. The top level algorithm is shown in Fig. 7.8.

Selecting modes for tasks

We select modes for tasks by generating legal mode combinations of tasks that satisfy the MDG. Note that the MDG used for a schedule may be a mix of resource dependency and task dependency, which represent time-invariant and time-variant dependency of resources. For example, in Fig. 7.9, the sub-graph in the dashed box represents *resource* dependency, whereas the rest of the graph shows the *task* dependency. We can still use the algorithm introduced in last section to generate legal mode combinations of tasks. Once a legal mode combination is determined, we can obtain a new schedule since the operation delay of tasks become known under their selected modes and under their co-activation dependency. We check timing constraints for the new schedule. If it fails, we generate another legal mode combinations and check again; if it passes, we use the mode combination for mode selection of idle intervals.

Scenario	Task sequence	Cost			Relative energy (simple / greedy / modesel)
		simple	greedy	modesel	
A	CAM/MOV/SCI	19002	18442	17935	100% /97.0%/93.4%
B	MOV/CAM/SCI	16381	15013	14667	100% /91.6%/89.5%
C	CAM/SCI/MOV	20294	19505	19014	100%/96.1%/93.6%

Mission tasks: CAM: shoot images; MOV: move to another location; SCI: perform scientific experiments.
Approaches: simple: assume two modes; greedy: greedily voltage scaling; modesel: our algorithm.

Figure 7.10: Comparison among different working scenarios.

Selecting modes for idle intervals

On each resource, overhead may incur on mode changes. We find a set of modes for each idle interval such that the time overhead of the mode changes is less than the length of the idle interval. We sort the modes in each set in an ascending order, and use modes in these sets to select modes for idle intervals and check power constraints. We treat overhead as additional tasks to the schedule we obtained. We characterize those overhead tasks with time and average power, which can be derived from time and energy information. We decompose the new schedule into *time intervals* such that within each time interval there is no task event (start or end event). The decomposition is done in the following way: We find the start and end events of all tasks. All the events cut the time axis into non-overlapping segments. Each segment forms a time interval. We check system power constraints in each time interval. If the schedule fails power constraints, we attempt a mode change on resources that currently have an idle interval, and check power constraints again. If all the modes fail the power constraints, we backtrack to the previous time interval. If we backtrack to the beginning of the schedule and still cannot find feasible modes, we attempt the next legal mode combination and select modes for idle intervals again.

7.5 Experimental Results

We apply our algorithm to an example based on the Mars rover [66]. The rover travels on the surface of Mars to perform scientific experiments and shoot images. Its resources consist of a camera (CAM), scientific devices (SCI), a radio-frequency modem (RF), a microprocessor (PPC), a hazard detector (HAZ), driving motors (DRV) and steering motors (STR). CAM takes a picture, sends the picture data to PPC for processing, PPC outputs to RF, and then the rover moves to another location (HAZ, DRV, STR) to perform scientific experiments (SCI, PPC, RF).

PPC can work at a number of different clock rates (with a full speed of 500MHz) and can be set to doze, nap or sleep modes. RF can be in rx only mode, tx-rx mode and sleep modes. The other resources have only two modes each, on and off. Mode-change overhead is significant for some resources. Due to the low temperature on Mars, DRV must be pre-heated for some time before turned on. Similar reason applies to STR, RF, and SCI. The inter-resource relationships are shown in Fig. 7.9. For example, when HAZ is working, neither DRV nor STR should be working. RF may be in tx-rx mode if and only if the processor is operating.

Fig. 7.11 shows a feasible mode schedule, in both time view and power view. Task $ppc2$ on PPC cannot be further slowed down because PPC and RF must be co-active. If PPC is greedily slowed down, it will violate max power constraint during the interval 500 - 560. Task $drv1$, $haz1$ and $str1$ are not overlapped due to the system requirement specified in the mode dependency graph. STR and SCI need significant time to pre-heat, which is adequately considered (the light gray areas in their tracks). Idle interval between $rf1$ and $rf2$ on RF is set to rx only rather than off because the timing overhead of mode changes (including pre-heating) is larger than the length of the interval. The idle interval before $rf1$ is set to rx only for the same reason.

We compared our algorithm with two other approaches: approach one assumes only two modes, on and offapproach two greedily applies voltage scaling technique

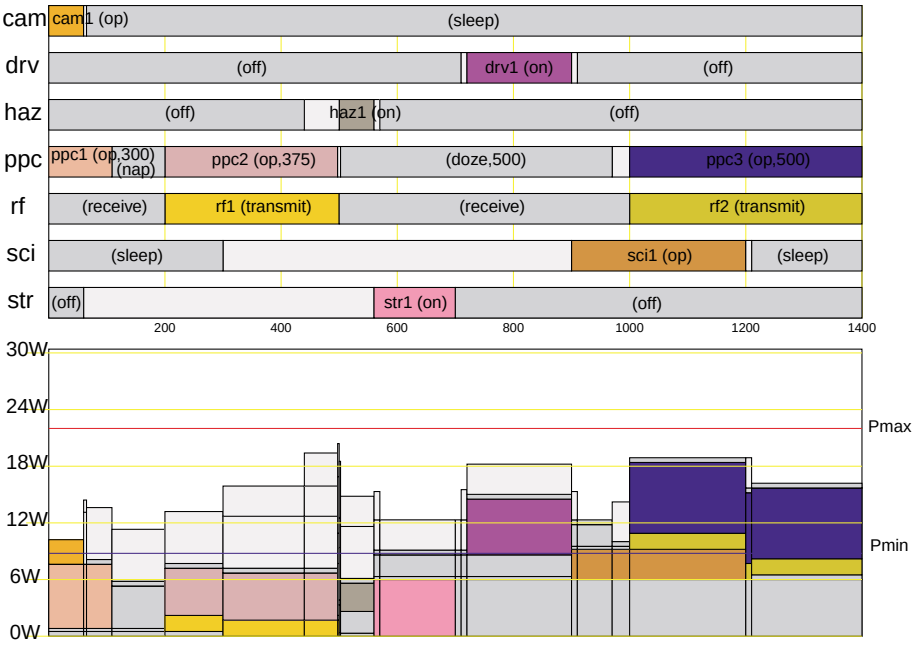


Figure 7.11: A mode schedule for microrover.

whenever possible (we allow power constraint violation in this approach). The results are shown in Fig. 7.10. Approach one gives the worst results because it never utilizes available modes. Approach two is better than approach one since it saves energy by applying voltage scaling technique, but its greediness pays the cost since its saving by slowing down the processor is more than offset by the extra energy consumed on the RF modem. And in all the scenarios, approach two violates max power constraint. Our algorithm gives the best results because we utilize multiple modes of resources and apply voltage scaling on the processor. At the same time, we avoid extra energy cost on RF by identifying co-activation dependency between the two resources and performing mode selection to find the feasible solution.

7.6 Chapter Summary

This chapter presents a method for capturing mode dependency and an algorithm for mode selection in power-aware embedded systems. The mode dependency graph introduced in this chapter enables legal combinations of modes to be systematically derived. Today's designers perform this task manually. However, as components offer increasingly sophisticated modes for power management, while at the same time imposing even more restrictions on mode changes, the complexity will grow quickly beyond what humans can handle. Our MDG represents a structured approach to controlling the complexity of power management. We also present a search algorithm that takes advantage of the MDG. By considering power/timing constraints and overhead on transitions, this technique gives designers more confidence in the feasibility of the synthesized results in real-life applications. Furthermore, our algorithm incorporates heuristic ordering to optimize for the energy cost of the solution, and it shows realistic, *system-level* improvements over previous techniques that either do not handle constraints or multiple components.

Chapter 8

Topology Selection

The trend towards distributed, networked embedded systems is changing the way power should be managed. Power consumed by bus and network interfaces now matches if not surpasses that of the CPU and is thus becoming a prime candidate for reduction. This chapter explores the energy-efficient bus topologies as a new technique for global power optimization of embedded systems that are interconnected by high-speed serial network-like busses such as FireWire and a new generation of SoC busses. Our grammar-based representation for these networks enables the modeling and facilitates selection of energy-efficient bus topology. Experimental results show 15-20% energy saving on the network interfaces without sacrificing system performance.

8.1 Introduction

A recent trend in power-aware designs is *communication centric* power management. In both embedded systems and system-on-chip (SoC) architectures, much of the research work in the past decade has gone into making the CPU very power efficient, and the CPU is now consuming a much smaller fraction of the system power. At the same time, bus and network interfaces are consuming the same if not more power. Higher

level integration helps alleviate the situation somewhat, but even processors with built-in network interfaces often require two supply voltages: a lower voltage for the core, and a higher voltage for the off-chip I/O. System-on-chip architectures will also face similar issues, as IP components are increasingly being integrated using on-chip networks for power and modularity advantages.

Communication-centric power management schemes can be divided into *custom protocols* vs. *standard protocols*. Custom protocols that utilize application-specific coding schemes [65, 47, 38, 24, 6], custom bus voltages [39], or custom bus segmentation schemes [76, 36, 16] can potentially achieve much better energy efficiency, but they are applicable mainly to closed systems. Most embedded systems and IP components must be interoperable with existing standards, and this limits the types of optimization possible. We do not attempt to propose a new standard to compete against the more established ones [7, 21, 22, 31, 33]; instead, it is intended to demonstrate how an existing standard can incorporate energy efficient optimizations. Some of the most important parameters include communication speed and bus topology. We investigate topology selection for FireWire, a hot-pluggable, low-power, high-speed serial bus that can support real-time streaming (isochronous) and asynchronous transfer modes. It is widely available on many embedded systems and computers today.

FireWire requires a tree topology. Furthermore, each FireWire component has a limited number of ports and a maximum transfer speed available. Our approach to achieve energy reduction is a grammar-driven, constraint-based searching process for low energy network topology. The advantages are: a) the formal method is a systematic way of modeling and generating topologies; b) our technique is extensible to other buses/networks and is beneficial to system-on-chip design with on-chip networks; c) it is orthogonal to most of the existing CPU-centric power management techniques, thus enabling additive energy savings by combining our techniques with existing ones. Our experimental results show up to 15% to 20% energy savings for network interfaces

without sacrificing system performance.

This chapter is organized as follows. Section 8.2 provides background information on FireWire and reviews related work. Section 8.3 presents a formal problem formulation, while Section 8.4 describes the algorithms we used to select optimal tree topologies. We discuss the experimental results in Section 8.5.

8.2 Background and Related Work

8.2.1 FireWire Bus

FireWire (IEEE1394) [8] is a high-speed serial bus standard. 1394a currently supports transmission speeds up to 400Mbps, and the new 1394b standard [5] will support transmission speeds of 800Mbps and 1600Mbps. FireWire was designed to connect a computer to peripherals such as hard disks, scanners, and consumer electronics such as video cameras. It is now widely available on many computers, set-top boxes, and embedded systems in automotive and aerospace domains. FireWire supports two data transfer types: asynchronous and isochronous transfer modes. Asynchronous mode guarantees the data delivery with acknowledgment. Isochronous mode guarantees data bandwidth without acknowledgment, and it is suitable for real-time streams such as video.

FireWire is hot-pluggable and can connect up to 63 devices. 1394a cables can run as long as 4.5 meters, and packets can take up to 16 hops for a maximum total distance of 72 meters. Future standard extends the single hop distance to up to 100 meters and use fiber optics as the physical medium. When a new node is attached to the bus, or an existing node is unplugged, the bus will go through bus reset. First, a root will be elected, followed by tree identification and self identification processes, after which the new topology map and speed map is broadcast to every node. Unlike the Universal Serial Bus (USB), which is host-based, FireWire is peer-to-peer.

FireWire imposes a number of restrictions. First, the network must be acyclic. This implies that there is a unique path between any pair of communicating nodes. Second, all intermediate nodes on the path must be powered on (at least the physical layer controller) to act as repeaters. Third, all the intermediate nodes must support the transfer speed of the communicating nodes, otherwise the transaction cannot be started. Fourth, the fan-out of each node is constrained by the number of ports available on the physical interface.

8.2.2 Power Management with FireWire

Power management opportunities with a standard protocol like FireWire are at higher level than most previous works. Circuit-level bus voltage scaling techniques, including [39, 70], which make the bus voltage and frequency track the bus traffic, or voltage swing reduction [55], would not be applicable due to interoperability reasons. Bus coding that minimizing the transition activities on buses [65, 47, 38, 24, 6] would not be applicable, either. Buses segmentation to reduce bus load and improve latency [76, 36, 16] may be applicable in principle, but they must be adapted to the specific capabilities of the bus standard. Our technique is similar to bus segmentation in the sense that both try to localize the bus traffic so that the high-cost global bus activities are minimized. While traditional bus segmentation techniques mainly partition and cluster the bus nodes into segments, our approach works with the constraints imposed by the bus standard on the topology, port count, and transfer speed. To accomplish this, we model the legal topologies using a tree grammar, and we use the constraints to prune the search space. We present an algorithm that finds a topology that minimizes total energy consumption for the same communication traffic. The experimental results are validated using a FireWire snooper.

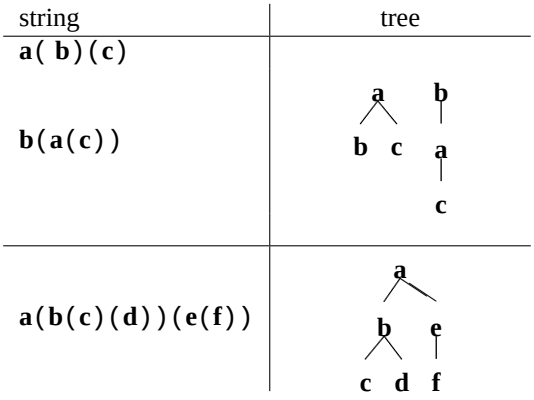


Figure 8.1: Examples of tree strings.

8.3 Problem Formulation

We generate tree topologies for FireWire by incrementally attaching new nodes to existing trees. We have developed a formal representation for modeling trees and generating tree topologies. In this section we give several definitions, followed by the cost function and our problem statement.

8.3.1 Definitions

Definition 21 (Node $\mathbf{u} \in U$) A node **u** is a component in the system that has bus interfaces ready to connect to other components. $p_{\mathbf{u}}$ is the number of ports available for **u**. $S_{\mathbf{u}}$ is a finite set of speeds that node **u** can work at.

Definition 22 (Tree) A tree is a connected component $C \subseteq U$ with exactly $|C|-1$ undirected edges.

Definition 23 (Transaction $\tau \in \Gamma$) A transaction $\tau = (\mathbf{u}_1, \mathbf{u}_2, s, w)$ is a data transfer behavior between two nodes $\mathbf{u}_1$ and $\mathbf{u}_2$ at the transfer speed s with non-zero workload w, where $s \in S_{\mathbf{u}_1} \cap S_{\mathbf{u}_2}$ and w is the amount of data (in byte) transfered.

We require that the transactions in the system be peer-to-peer. Multicast or broadcast transactions are not considered.

Definition 24 (Tree string t) A tree string is a string representation of a tree. It is obtained by in-order traversal of the tree. The root of the tree is traversed first, then recursively each child is traversed. For example, in Fig. 8.1, the string **a(b)(c)** represents a tree of three nodes, with **a** the root node and **b** and **c** leaf nodes. A matched pair of parentheses with the substring inside represents a subtree. The string **a(b(c)(d))(e(f))** represents a tree of six nodes. **c** and **d** are two subtrees (also leaf nodes) of **b**, and **b(c)(d)** and **e(f)** are two subtrees of **a**.

Definition 25 (Tree grammar G) Let Σ be an alphabet $\Sigma = \{\mathbf{u}|\mathbf{u}\in U\}\cup\{\mathbf{(},\ \mathbf{)}\}$, and a node **u** is denoted by a lower-case Roman letter. A tree is represented by a tree string t that can be generated from grammar $G = (V,\Sigma,P,S)$, where $V = \{B,E\}$ is a set of *variables*, S is a start symbol, P is a set of productions $V \rightarrow V\cup\Sigma$:

$$E \rightarrow \mathbf{u}$$
$$B \rightarrow \mathbf{(}E\mathbf{)}$$
$$E \rightarrow EB$$
$$S \rightarrow E$$

and if a node **u** appears in t, it appears exactly once.

Definition 26 (Tree language L) A language $L(\Sigma) = \{t|t$ is in Σ^* and $S \Rightarrow t\}$ is a set of tree strings generated by grammar G. We also use $L(v) = \{t|t$ is in Σ^* and $v \Rightarrow t\}$ to denote the set of strings generated with the start symbol $v \in V$, and $L(v^*) = \{t^*|t$ is in Σ^* and $v \Rightarrow t\}$ to denote the set of strings that has zero or one or more concatenated substrings each of which is generated with a start symbol $v \in V$.

Let $|t|$ represent the length of the tree string t. It is easy to see that for a tree containing n nodes, $|t| = 3n-2$.

A tree topology can be represented by multiple tree strings. For example, **a(b)(c)** and **b(a(c))** in Figure 8.1 represent the identical topology with different roots. Even with the same root, tree string **a(b)(c)** and **a(c)(b)** represent the same tree. Since any node (capable of bus management) on a FireWire bus can be the root, we can pick one node as the root and order the rest so that we are able to obtain a canonical form of a tree string.

Definition 27 (Transforming function H) A transforming function H converts a tree string to its canonical form by the means of in-order traversal with sorting of labels. The canonical form of a tree string t is: $\forall$**u** in t, **u** and its all children are sorted in a lexicographical order. Tree string t_1=**a(b(c)(d))(e(f))** in Figure 8.1 is in its canonical form. Tree string t_2=**a(e(f))(b(c)(d))** is not in its canonical form since **a** and its children **e** and **b** are not sorted. Thus we have $t_1 = H(t_2)$.

New trees can be formed by adding a node x to an existing tree. The node can be either attached as a leaf node or inserted as a non-leaf node. We define a growing pattern $F(t,x)$ to help incrementally generate larger trees from smaller ones.

Definition 28 (Growing function F) $L(\Sigma \cup \{x\}) = L(\Sigma) \cdot F(t,x)$, for all $t \in L(\Sigma)$. Tree strings in $L(\Sigma \cup \{x\})$ can be derived from trees in $L(\Sigma)$ according to the following rules:

$$F(t,x) = \begin{cases} \mathsf{d}(x) & \text{if } t = \mathsf{d}, \\ \mathsf{d}(x)(\beta)\gamma \cup \mathsf{d}(x(\beta))\gamma \cup & \\ \mathsf{d}(F(\beta,x))\gamma \cup \mathsf{d}(\beta)F'(\gamma,x) & \text{if } t = \mathsf{d}(\beta)\gamma. \end{cases} \tag{8.1}$$

$$F'(\alpha,x) = \begin{cases} \emptyset & \text{if } \alpha = \varepsilon, \\ (F(\beta,x))\gamma \cup (\beta)F'(\gamma,x) & \text{if } \alpha = (\beta)\gamma. \end{cases} \tag{8.2}$$

where $\mathsf{d} \in U$ represents the root of tree t, $\beta \in L(E)$ and $\gamma \in L(B^*)$.

Definition 29 (Tree string set T) A tree string set T for a node set U is a set of tree strings generated by grammar G and:

1) $\forall t \in T, \forall u \in U, u$ is in t,

2) $\forall t \in T, t = d_0 D, d_0 \in U, D \in L(B^*)$,

3) $\forall t_1, t_2 \in T$, if $t_1 \neq t_2$, then $H(t_1) \neq H(t_2)$, and

4) for any tree string $t = d_0 D, d_0 \in U, D \in L(B^*)$, $|t| = 3|U| - 2$,
$\exists t' \in T, H(t) = H(t')$.

In other words, each tree string in T contains all nodes in U. All the tree strings have the same root node d_0. No two tree strings in T have the same canonical representation. Tree set T represents a complete set for all the tree topologies for the node set U.

We assume that all the nodes in U are connected to form a single tree topology. A forest consisting of multiple trees is not allowed.

Lemma 8 (Tree generation) Given a tree string set T for a node set U, a new tree string set T' for the node set of $U \cup \{x\}$ is derived from T without producing identical topologies by applying growing function F to each tree in T: $T' = T \cdot F(t, x)$, for all $t \in T$.

Definition 30 (Port count constraint) A tree can be represented in the form of $\mathbf{d}B^k$, where $\mathbf{d}$ represents the root of the tree, and $B^k (B \in L(B))$ represents all of its k subtrees. The port count constraint is: $\forall u \in U$,

$$\begin{cases} p_u \geq k+1 & \text{if } p \text{ is a non-root node,} \\ p_u \geq k & \text{if } p \text{ is the root node.} \end{cases} \tag{8.3}$$

where p_u is the port count for node u.

For example, in Figure 8.1, the tree string $\mathbf{a(b(c)(d))(e(f))}$ satisfies (8.3) if $f_a, f_e \geq 2$, $f_b \geq 3$ and $f_c, f_d, f_f \geq 1$.

Corollary 1 (Connectivity condition) Given a node set U of n nodes, a tree topology that connects all the nodes exists, iff

$$\sum_{u \in U} p_u \geq 2n - 2 \tag{8.4}$$

where p_u is the port count of node u.

A tree is *legal* if every node in U satisfies the port count constraint (8.3) and connectivity condition (8.4).

8.3.2 Cost Function

Given a transaction $\tau = (u_\tau, v_\tau, s_\tau, w_\tau)$, all the nodes $u \in U$ can be categorized into three sets: M_t, M_r, and M_i. $M_t = \{u_\tau, v_\tau\}$ consists of communicating nodes. M_r consists of all the nodes that repeat the transaction τ on the routing path. M_i consists of the nodes not involved in the transaction τ. We say the working modes m_u for the node u in the above sets each are transferring, repeating, and idle, respectively.

For a given node u, the power function P is a function of the port number p_u and working mode m_u, denoted as $P(p_u, m_u)$. Power function can be a lookup table whose data entries come from manufacture's data sheets [67].

We define the power function of a transaction τ and a tree t as:

$$P(\tau, t) = \sum_{u \in M_t} P(p_u, m_u) + \sum_{u \in M_r} P(p_u, m_u) \sum_{u \in M_i} P(p_u, m_u) \tag{8.5}$$

Power function $P(\tau, t)$ represents the total bus power of the whole systems during the transaction τ. It consists of power of nodes involved the transaction (both transferring and repeating node) and power of idle nodes.

For a transaction τ, *Effective transaction time* is defined as:

$$D_\tau = \frac{w}{s}. \tag{8.6}$$

where w is the workload and s is the transmission speed.

Note that effective transaction time may not be equal to the actual time to complete a transaction. Consider two transactions start at the same time, both transferring data at the same speed and both taking one minute to complete. Assume they equally share the total bandwidth, the effective transaction time for each transaction is only half a minute.

During a given time period D, we suppose there are k transaction instances $\{\tau_i\}(i = 1,\ldots,k)$. The total effective transaction time is:

$$D_\tau = \sum_i D_{\tau_i}. \tag{8.7}$$

We define *utilization* of the transaction τ is:

$$\lambda_\tau = \frac{D_\tau}{D}. \tag{8.8}$$

Finally for a given tree string t, we define our cost function as:

$$C = \sum_{\tau\in\Gamma} P(\tau,t)\lambda_\tau \tag{8.9}$$

Cost C represents the average energy consumption on the bus in unit time. However it does not include the energy consumption when the bus is completely idle (no transaction occurs).

8.3.3 Problem Statement

Given a tree t and a set of transactions Γ, the tree is a *feasible* one if it satisfies the speed constraint:

$$\forall\ \tau(u_\tau, v_\tau, s_\tau, w_\tau) \in \Gamma \text{ and } \forall\ x \in M_r,\ s_\tau \in S_x. \tag{8.10}$$

That is, for a transaction, all the intermediate nodes on a routing path should support the transfer speed. We aim to find trees that has the minimum cost defined by (8.9). The input to the problem is a set of node V and a set of transaction Γ. The output of the problem is a tree (or a set of trees) with minimum cost.

In case the input node set U does not satisfies the connectivity condition (8.4), we add *hubs* to connect the nodes so that the connectivity condition is satisfied. A hub is a special node that can repeat transactions but cannot be a peer node in a transaction. Several types of hubs are available, which are differentiated by their port count. The more ports a hub has, the more power it consumes when repeating packets. Part of the topology selection problem is to select different hub types for energy optimality.

8.4 Algorithm

Tree topologies are incrementally generated using our grammar-based growing function. In this section we present the tree generation algorithm and the top level search algorithm. A brief discussion on complexity shows that asymptotically our algorithm generate much fewer trees than exhaustive approach, and in practice, our technique produce even much fewer trees by applying system-level constraints.

8.4.1 Approach

We take an incremental approach to obtaining the tree set of $k+1$ nodes from a tree set of k nodes. We use our growing function F to add a node to an existing tree either as a leaf node or as a non-leaf node. At each incremental step, if a tree topology fails to satisfy the port count constraint or the transfer speed constraint, it will not be included into the tree set. After obtaining a tree set for all the nodes, we calculate cost for each tree to search for optimal topologies.

```
TREEGEN(V, Γ, h)
0   # input : node set U, transaction set Γ, hub type h
1   # output: tree set T
2   # Preprocess: add hub nodes if necessary
3   V' ← preprocess(V,h), #sort nodes in decreasing order by their p_u.
4   for each v in U' { p[v] ← p_u } # p[v]: port count of V.
5   v ← pop up the first node in U'
6   T ← {u}
7   while U' not empty {
8       u ← pop up the first node in U'
9       T' ← T
10      for each tree t in T {
11          for each node v in t {
12              T_l ← ADDASLEAF(t,u,Γ)
13              T_b ← ADDASBRANCH(t,u,Γ)
14              T' ← T_l ∪ T_b
15          }
16      }
17      T ← T'
18  }
19  return T
```

Figure 8.2: The tree enumeration algorithm.

8.4.2 Algorithms

The tree generation algorithm is shown in Figure 8.2. The inputs to the algorithm are a node set U and hub type h. The output of the algorithm is a tree set containing all the feasible trees. In the pre-process procedure in Line 3, we check whether the node set U satisfies the connectivity condition (8.4). If port count is not enough, we add an adequate number of hubs of type h into the node set, thus forming a new node set U'. We also sort the nodes in U' by their port counts to facilitate the tree generation described below. Array $p[n]$ (Line 4) keeps the port count information of all nodes in U' during the process of tree generation. Lines 5–6 gets the first node in U' and initialize the tree set T. The while loop (Lines 7–18) incrementally generates new trees and expands tree set. Two main steps are ADDASLEAF() and ADDASBRANCH() which add a new node to the existing tree set as a leaf node and as a non-leaf node, respectively.

```
ADDASLEAF(t, x, Θ)
    # input : tree t, node x, transaction set Γ
    # output: tree set T_l
    T_l ← ∅
    ptr ← 0
    while ptr < len(t) {
        while t[ptr] ∉ D { ptr ← ptr + 1} # find next node id
        if p[t[ptr]] > 0 { # if port available
            T_sub ← Subtree(t[ptr]) # T_sub: a set of subtrees of t[ptr]
            insertx(T_sub,'(x)') # so that elements in T_sub are sorted
            t' ← join(T_sub) # concatenate elements in T_sub into a string
            t'' ← insertSub(t,t') # substitute t[ptr]'s subtrees for t'
            updatePort(p) # update port count information
            tag ← 1
            for each τ ∈ Θ {
                if not checkSpeed(t'',τ) {
                    tag ← 0; break }
            }
            if tag == 1 { T_l ← T_l ∪ {t''} }
        }
        ptr ← ptr + 1
    }
    return T_l
```

Figure 8.3: The ADDASLEAF routine.

```
MINTREE(V, Θ, H)
    # input : node set V, transaction set Θ, hub type set H
    # output: optimal tree set minTreeSet, minimum cost minCost
    minTreeSet ← ∅
    minCost ← ∞
    for each h in H {
        T ← TREEGEN(V,h)
        for each t in T {
            cost ← getCost(t,Θ)
            if cost < minCost {
                minCost ← cost; minTreeSet ← {(t,h)}
            } else if cost == minCost {
                minTreeSet ← minTreeSet ∪ {(t,h)}
            }
        }
    }
    if minCost < ∞ {print minCost, minTreeSet }
    else { print "no solution found"}
```

Figure 8.4: The top level topology selection algorithm.

Figure 8.3 shows the procedure ADDASLEAF(). When adding a new node x to an existing tree t as a leaf node, we try attaching x to each node if it has a port available. In the string of tree t, we insert (x) after a node u to the right position so that the new tree remains in its canonical form. We identify the routing path between two communicating nodes u and v, check speed constraints for every intermediate nodes (if any), and return whether the tree satisfies the speed constraint. If for all transactions, the tree satisfies speed constraints, we append it to the tree set.

The other step, similar to ADDASLEAF(), is to add x as a branch node. A connection between a node u and one of its subtrees is identified. x is inserted as the child of u while the subtree as the child of x. Thus x becomes a non-leaf node. We repeat this for all the subtrees of u. The procedure ADDASBRANCH() is implemented similarly to ADDASLEAF() as string manipulation and it not shown.

A top level algorithm is shown in Figure 8.4. We assume the connectivity condition (8.4) is not satisfied thus we try different types of hubs in the outmost loop (Lines 5 and 15). Otherwise the loop can be safely removed. Line 6 generates all feasible trees and stores them in set T. In the loop of Lines 11–19, we check to see whether the speed constraint is satisfied. If yes, we then calculate its cost and save it if it is minimum cost. Finally we output the optimal tree set with hub type or no solution message if all topologies fails to satisfy the constraints.

8.4.3 Complexity

The complexity of an exhaustive approach is prohibitive. Given a node set U of n nodes, we can permute the nodes and obtain $n!$ strings, each consisting of n nodes. For each string, we need to add $n-1$ pairs of parenthesis to form a tree string. For each parentheses pair, we have $n-1$ locations to add, and adding parentheses pairs is independent with each other. Thus we have 2^{n-1} of ways to add $n-1$ pairs of parentheses. Altogether, we can obtain $n!2^{n-1}$ trees from a node set of n nodes. Note that among

Port/mode	Transfer	Repeat	Idle
1	158.4	138.6	125.3
2	234.3	217.7	174.7
3	379.5	320.1	247.5
4	676.5	498.3	412.5
6	924.0	673.2	541.2

Table 8.1: Power data of FireWire interface (in mW).

those trees, there are trees that topologically identical but differ in root nodes, trees that are not in their canonical forms, and trees that do not satisfy constraints.

Our algorithm assumes a node to be the root node and trees differ only in the root will not be repetitively generated. Our algorithm generates tree strings in their canonical forms and does not generated topologically identical tree strings.

In the ADDASLEAF() routine, the if-branch (Lines 8–18) produces at most k new strings (k is the number of nodes in current tree t). ADDASBRANCH() routine produces at most $(k-1)$ new strings. Thus we obtains at most $(2k-1)$ tree strings for a tree of $(k+1)$ nodes. Theoretically, our algorithm may produce at most $(2n-3)!!$ patterns for a node set of size n, which sets a very loose upper bound of generated tree strings. This is already asymptotically smaller than the exhaustive approach. In reality, our algorithm generates much fewer trees since we apply constraints at each incremental step. This greatly reduces the generated trees in that step and avoids fast growing of trees in the succeeding steps. For example, when $n = 8$, theoretically the exhaustive approach produces 5160960 trees and our algorithm may produce at most 135135 strings, only 2.6% of the former approach. In reality, we only generated as few as 90 trees (see Section 8.5), due to the constraints we applied.

8.5 Experimental Results

We apply our algorithm to two FireWire bus examples. We use FireBug [9], a software bus snooping tool, to monitor the bus traffic and obtain the workload information. In

Device	Max speed(Mbps)	port #
Mac1	400	2
Mac2	400	2
PC1	400	2
HD1	200	2
Cam	100	1
iBot1	200	1
ibot2	200	1
Hub	400	3/4/6

Table 8.2: A list of FireWire devices

Trans.	u1	u2	Speed(Mb/s)	Workload (x1000Mb)
1	Mac1	HD1	200	13
2	Mac1	PC1	400	25
3	Mac1	Cam	100	80
4	Mac1	iBot1	200	46
5	Mac2	HD1	200	5
6	PC1	iBot2	200	46

Table 8.3: A list of transactions

Hub type	$p = 3$	$p = 4$	$p = 6$
# of trees	90	269	376
MaxCost	270.6	306.2	338.9
MinCost	213.2	267.5	290.8
diff(%)	12.2	14.5	16.6
# of optimal trees	4	1	1

Table 8.4: Experiment results for Example I (eight nodes).

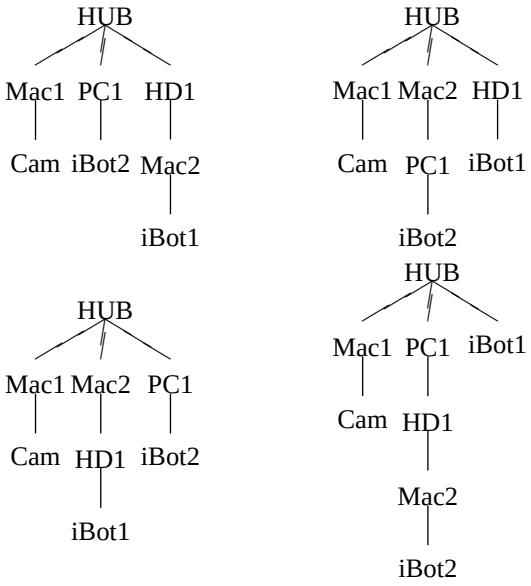


Figure 8.5: Example I: $p = 3$, four trees found.

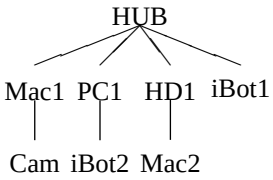


Figure 8.6: Example I: $p = 4$, one tree found.

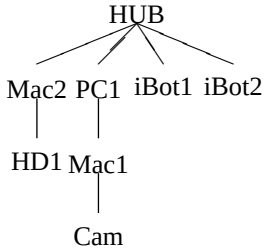


Figure 8.7: Example I: $p = 6$, one tree found.

the first example we have eight nodes to be connected. The second example has the similar setup but in a larger scale, which makes it almost impossible for the exhaustive approach to find out a solution in a practical time period. Our algorithm generates optimal tree sets efficiently. Our experimental results show that the optimal solutions we found save up to 15%–20% energy compared to an arbitrarily generated topology. Furthermore, workload balanceness and hub types have perceivable influences on energy cost. We have built a web-based tool to facilitate the user to interact with the core algorithm on the server side.

Example I

We have seven devices to be connected with FireWire bus interfaces, as listed in table 2: Mac1 and Mac2 are two desktop Mac computers, PC1 is a notebook computer, HD1 is a FireWire hard drives, Cam is a Camcorder, iBot1 and iBot2 are two web cameras. A hub is added to the device list in order to satisfy the connectivity condition.

We use FireBug to capture the workload information. FireBug can keep track of all the activity on the FireWire bus and report to the user the desired events by filtering out the irrelevant ones. We first arbitrarily interconnect all the devices and turn on FireBug to mornitor and record the traffic on the bus, and extract transaction-related information from FireBug log file. For example, we obtain the nodes involved in a transaction, the data transfer speed and the number of packets transferred. Then we obtain the transaction table shown in Table 8.3.

Table 8.4 shows the experiment results. Although this experiment looks simple, to find the optimal solution is not trivial. Exhaustive enumeration will produce 5,160,960 trees (see the last section). Our algorithm shrinks the tree set sizes down to less than 400 (first line of Table 8.4) using our grammar-driven tree generation.

MaxCost and *MinCost* are the maximum and minimum cost value for all generated feasible trees. In three cases ($f_n = 3, 4, 6$), the differences between *MinCost* and

MaxCost are ranging from 12.2% to 16.6%, representing the potential energy savings by selecting the trees with $MinCost$. It is interesting to see that the more ports the hub has, the more energy the tree consumes. The reason is that the hub with more ports consumes more energy to repeat packets. Therefore for this example, a three-port hub is the optimal solution.

Figs. 8.5, 8.6, and 8.7 show the optimal tree sets when using hubs of three, four, and six ports, respectively. When using a three-port hub, four trees are found (see Fig. 8.5). When using a six-port hub, only four ports of the hub are used. This is because for some transactions, it costs less when the two peer nodes are directly connected (if possible) instead of going through a hub. Trees in Fig. 8.6 and Fig. 8.7 are different even in both cases four ports are used. The reason is that different hub types in the two cases causes different energy consumption.

To see whether the potential energy savings are sensitive to the workload balanceness, we change the workload on transaction 3 (between *Mac*1 and *Cam*) and generate optimal topology for each workload value. Transaction 3 originally has the largest workload among all transactions. We change its workload value from the average of all transactions to positive infinity (disabling all other transaction). Fig. 8.8 shows the curve of the workload percentage vs. the potential energy savings. The workload percentage is the ratio between the workload of the selected transaction to the total workload of all transactions. The curve shows that the higher the workload percentage is, the higher the potential energy saving becomes. This implies that the more unbalanced the workload is, the more significant the potential energy saving becomes, by up to nearly 20% in this example.

Example II

We use three Mac computers, four FireWire hard drives, one printer, one scanner and one camcorder, totally ten devices. To satisfy the connectivity condition, we add three,

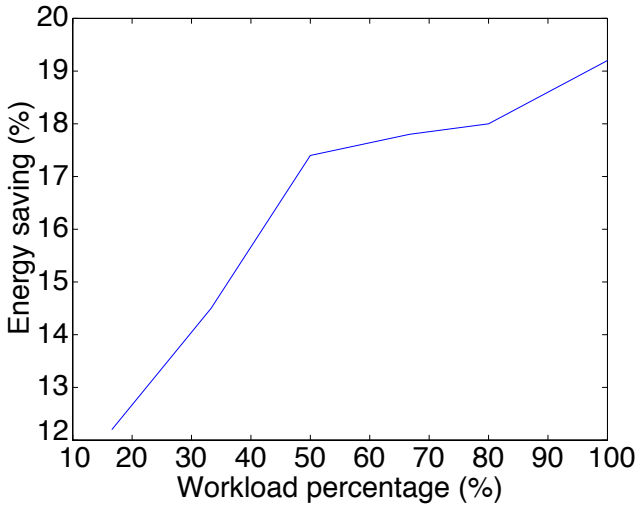


Figure 8.8: Workload balanceness vs. potential energy saving.

hub type	$p = 3$	$p = 4$	$p = 6$
# of hubs	3	2	1
# of total devices	13	12	11
# of trees	45761	17001	2013
MaxCost	332.8	304.4	270.4
MinCost	300.9	268.8	236.7
diff(%)	10.1	13.3	14.2
# of optimal trees	3	2	1

Table 8.5: The number of devices with different hub types.

two, and one hub when using three-port, four-port, six-port hubs, respectively (first two rows in Table 8.5). For the exhaustive approach, the problem of up to thirteen nodes becomes intractable in practice. Out algorithm generated highly compact tree sets (Row 3 of Table 8.5). Potential energy savings range from 10.1%–14.2%.

Note that in our cost function, we only consider the time periods when there is traffic on bus. When the bus is complete idle, part or all the bus nodes can be potentially disabled, resulting more energy savings. In the implementation of FireWire bus drivers, the link layer and above layers can be disabled for low power if no transaction is on the node. This is in contrast to our examples, where we assume all the layers are on all the time. Even for the physical layer controller, dynamic power management techniques can be applied to disable it when there is no traffic passing through it. All the above conditions are orthogonal to our techniques. It is conceivable that additive energy saving can be achieved by combining our technique with other power management techniques.

8.6 Chapter Summary

This chapter presents a method for optimizing peer-to-peer serial bus topology for energy reduction. To represent trees, we use a canonical string form that is both concise and easy to manipulate. We purpose an incremental approach to enumerating valid tree topologies. By applying a number of constraints in each enumeration step, we are able to obtain both complete and compact tree sets without producing redundant trees. We capture the bus workload information by monitoring the bus traffic and factor it into the cost function for energy optimization.

Workload distribution has an impact on the potential energy savings. The more imbalanced the workload is, the more energy saving opportunities exist. Hub type selection influences the optimal solution points due to variations in their individual power behavior.

Although we use FireWire bus to demonstrate the effectiveness of our technique, our approach is general enough to apply to many other tree-like architectures. As low power serial busses become more popular in computer systems and system-on-chip, we believe our technique can be applied to more applications and will show significant energy savings.

Current topology optimization is static, requiring the bus to reconfigure at least once to form an optimal topology. It is possible to construct a bus topology with redundant physical links while dynamically configuring it to form new tree topologies for performance, energy-saving, and fault tolerance.

Part V

Conclusion

Chapter 9

Conclusions and Future Work

This document presents the IMPACCT tool and methodology for system-level power management of power-aware embedded systems. The primary goal of the tool is to greatly expand the range of power/performance trade-offs, so that the system can most effectively adapt to the wide range of power availability in different operating scenarios. This can be accomplished by leveraging existing low-power and high-performance techniques, but a naive technique integration have led to incorrect results because important system-level properties were not properly considered. One of our contributions is precisely in modeling the important system-level dependencies including co-activation and inter-component modes. We have also developed power-aware scheduling and mode selection as two core tools for computing system-level power management policies. Our scheduler not only generates different schedules whose parallelism tracks the power availability, but also more aggressively increases the dynamic range by task motion while preserving timing and power constraints. Our mode selection methodology systematically exploits novel power management features in new components with a much richer set of power modes while considering all timing/power overhead associated with mode changes. All of these were made possible by our system-

level dependency modeling methodology. Also supported is a system-level simulation engine that coordinates the execution of heterogeneous models that can range from native code to detailed simulation models and even emulators. They comprise a powerful framework to aid the quick exploration and validation of power management decisions. We believe this work represents a major step towards a framework that will be able to effectively integrate the best power management techniques developed by others and by us.

We are currently pursuing several directions for future work. One ongoing project is to augment the library with a richer collection of components to include not only processor models but also more types of memory modules, peripheral devices, and battery models. To make our methodology practical and usable by engineers, we are also investigating automatic extraction techniques to reduce the effort in constructing the models needed as input to the IMPACCT tool. On scheduling, we are developing on-line, battery-aware algorithms under not only *power* constraints but also *energy* constraints. This will be supported by models for batteries and other energy sources [43, 51, 73]. Some initial work was recently proposed [46, 50], but we believe energy constraints must be considered in the context of the battery discharge and even recharge characteristics. Schedulers that are aware of battery discharge characteristics have been proposed [45, 51] but they do not treat power as constraints. On mode selection, it is being generalized to algorithm selection (e.g., between alternative image compression algorithms), which must be accompanied by data structure selection. Switching between algorithms and data structures will incur even larger timing and power overhead but the potential payoff will be tremendous. Together, we expect all these features will make IMPACCT a compelling tool for power-aware designs in the near future.

Bibliography

[1] The Alchemy Au1100 from AMD: Internet edge processor. http://www.alchemy-semi.com/product_info/au1100/index.html.

[2] INTEL ethernet PHYs/transceivers. http://developer.intel.com/design/network/-products/ethernet/linecard_ept.htm.

[3] INTEL XScale microarchitecture. http://developer.intel.com/design/intelxscale/.

[4] NASA/JPL's Mars Pathfinder home page. http://mars3.jpl.nasa.gov/MPF/-index0.html.

[5] 1394 Trade Association. P1394b draft standard for a high performance serial bus (high speed supplement). In *http://www.zayante.com/p1394b/drafts/p1394b1-33.pdf*, 2001.

[6] Y. Aghaghiri, F. Fallah, and M. Pedram. Irredundant address bus encoding for low power. In *Proc. of Int. Symposium on Low Power Electronics and Design*, pages 182–187, August 2001.

[7] V. Alliance. On chip bus attributes version 1. In *http://www.vsi.org/library/specs/summary.htm*, 1998.

[8] D. Anderson. *FireWire System Architecture*. MindShare Inc., Reading, Massachusetts, second edition, 1999.

[9] Apple Inc. Apple's firewire sdk 2.8.1. In *ftp://ftp.apple.com/developer/Development_Kits/FireWire_2.8.1_SDK.sit.bin*, 2000.

[10] N. K. Bambha, S. S. Bhattacharyya, J. Teich, and E. Zitzler. Hybrid global/local search strategies for dynamic voltage scaling in embedded multiprocessors. In *Proc. International Symposium on Hardware/Software Codesign*, pages 243–248, 2001.

[11] L. Benini and G. De Micheli. Networks on chips: a new soc paradigm. *IEEE Computer*, 35(1):70–78, Jan 2002.

[12] L. Benini, G. Paleologo, A. Bogliolo, and G. De Micheli. Policy optimization for dynamic power management. *IEEE Transactions on Computer Aided Design*, 18:813–833, June 1999.

[13] T. Burd and R. Brodersen. Design issues for dynamic voltage scaling. In *Proc. International Symposium on Low Power Electronics and Design*, pages 9–14, July 2000.

[14] A. Chandrakasan, S. Sheng, and R. Brodersen. Low-power CMOS digital design. *IEEE Journal of Solid-State Circuits*, 27(4):473–484, April 1992.

[15] L.-F. Chao, A. LaPough, and E. H.-M. Sha. Rotation scheduling: A loop pipelining algorithm. *IEEE Transactions on Computer Aided Design*, 16(3):229–239, March 1997.

[16] J. Chen, W. Jone, J. Wang, H.-I. Lu, and T. Chen. Segmented bus design for low-power systems. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, 7(1):25–29, March 1999.

[17] R. Cherabuddi, M. Bayoumi, and H. Krishnamurthy. A low power based system partitioning and binding technique for multi-chip module architectures. In *Proc. Proc. Great Lakes Symposium on VLSI*, pages 156–162, 1997.

[18] P. Chou and G. Borriello. Software scheduling in the co-synthesis of reactive real-time systems. In *Proc. Design Automation Conference*, pages 1–4, June 1994.

[19] P. Chou and G. Borriello. Interval scheduling: Fine grained code scheduling for embedded systems. In *Proc. Design Automation Conference*, pages 462–467, June 1995.

[20] E.-Y. Chung, L. Benini, and G. De Micheli. Dynamic power management using adaptive learning tree. In *Proc. International Conference on Computer-Aided Design*, pages 274–279, 1999.

[21] H. Consortium. Hypertransport I/O link specification 1.03. In *http://www.hypertransport.org/downloads/HT_IOLink_Spec.pdf*, 2001.

[22] W. J. Dally and B. Towles. Route packets, not wires: On-chip interconnection networks. In *Proc. of DAC*, pages 684–689, June 2001.

[23] P. Eles, A. Doboli, P. Pop, and Z. Peng. Scheduling with bus access optimization for distributed embedded systems. *IEEE Transactions on VLSI Systems*, 8(5):472–491, 2000.

[24] J. Henkel and H. Lekatsas. A^2BC: adaptive address bus coding for low power deep sub-micron designs. In *Proc. of the 38th Design Automation Conference*, pages 744–749, June 2001.

[25] I. Hong, D. Kirovski, G. Qi, M. Potkonjak, and M. B. Srivastava. Power optimization of variable voltage core-based systems. In *Proc. Design Automation Conference*, pages 176–181, June 1998.

[26] I. Hong, D. Kirovski, G. Qu, and M. Potkonjak. Power optimization of variable-voltage core-based systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(12):1702–1714, 1999.

[27] I. Hong, M. Potkonjak, and M. B. Srivastava. On-line scheduling of hard real-time tasks on variable voltage processor. In *Proc. International Conference on Computer-Aided Design*, pages 653–656, November 1998.

[28] I. Hong, G. Qu, M. Potkonjak, and M. Srivastavas. Synthesis techniques for low-power hard real-time systems on variable voltage processors. In *Proc. IEEE Real-Time Systems Symposium*, pages 178–187, December 1998.

[29] E. Huwang, F. Vahid, and Y.-C. Hsu. FSMD functional partitioning for low power. In *Proc. Design, Automation and Test in Europe*, pages 22–28, 1999.

[30] C.-H. Hwang and A. Wu. A predictive system shutdown method for energy saving of event-driven computation. In *Proc. 1997 Design Automation Conference*, November 1997.

[31] IBM. Coreconnect bus architecture. In *http://www.chips.ibm.com/products/coreconnect/index.html*, 1999.

[32] C. Im, H. Kim, and S. Ha. Dynamic voltage scaling technique for low-power multimedia applications using buffers. In *Proc. International Symposium on Low Power Electronics and Design*, August 2001.

[33] Intel. Third generation I/O architecuture. In *http://developer.intel.com/technology/3GIO*, 2001.

[34] T. Ishihara and H. Yasuura. Voltage scheduling problem for dynamically variable voltage processors. In *Proc. International Symposium on Low Power Electronics and Design*, pages 197–202, August 1998.

[35] M. Jacome, G. de Veciana, and C. Akturan. Resource constrained dataflow retiming heuristics for vliw asips. In *Proc. International Symposium on Hardware/Software Codesign*, pages 12–16, May 1999.

[36] J. Kim and A. El-Amawy. Performance and architectural features of segmented multiple bus system. In *Proc. of International Conference on Parallel Processing*, pages 154–161, 1999.

[37] P. V. Knudsen and J. Madsen. Integrating communication protocol selection with hardware/software codesign. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(8):1077–1095, August 1999.

[38] S. Komatsu, M. Ikeda, and K. Asada. Low power chip interface based on bus data encoding with adaptive code-book method. In *Proc. Ninth Great Lakes Symposium on VLSI*, pages 368–371, March 1999.

[39] L.-S. P. L. Shang and N. Jha. Power-efficient interconnection networks: Dynamic voltage scaling with links. *Computer Architecture Letters*, 1(2), May 2002.

[40] K. Lahiri, A. Raghunathan, and G. Lakshminarayana. LOTTERYBUS: a new high-performance communication architecture for system-on-chip designs. In *Proc. Design Automation Conference*, pages 15–20, June 2001.

[41] K. Lalgudi and M. Papaefthymiou. Fixed-phase retiming for low power design. In *Proc. International Symposium on Low Power Electronics and Design*, pages 259–264, August 1996.

[42] C. Leiserson and J. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6(1):5–35, 1990.

[43] H. Linden. *Handbook of Batteries*. McGraw-Hill, 1995.

[44] J. Liu, P. H. Chou, N. Bagherzadeh, and F. Kurdahi. Power-aware scheduling under timing constraints for mission-critical embedded systems. In *Proc. Design Automation Conference*, pages 840–845, June 2001.

[45] J. Luo and N. K. Jha. Battery-aware static scheduling for distributed real-time embedded systems. In *Proc. Design Automation Conference*, pages 444–449, June 2001.

[46] T.-L. Ma and K. Shin. A user-customizable energy-adaptive combined static/dynamic scheduler for mobile applications. In *Proceedings 21st IEEE Real-Time Systems Symposium*, pages 227–236, November 2000.

[47] E. Musoll, T. Lang, and J. Cortadella. Working-zone encoding for reducing the energy in microprocessor address bueses. *IEEE Trans. on VLSI Systems*, 6(4):568–572, December 1998.

[48] T. Okuma, T. Ishihara, and H. Yasuura. Real-time task scheduling for a variable voltage processor. In *Proc. International Symposium on System Synthesis*, pages 24–29, November 1999.

[49] R. Ortega and G. Borriello. Communication synthesis for distributed embedded systems. In *Proc. International Conference on Computer-Aided Design*, pages 437–444, 1998.

[50] A. Parikh, M. Kandemir, N. Vijaykrishnan, and M. Irwin. Energy-aware instruction scheduling. In *Proc. International Conference on High Performance Computing*, pages 335–344, December 2000.

[51] M. Pedram, C.-Y. Tsui, and Q. Wu. An integrated battery-hardware model for portable electronics. In *Proc. Asia and South Pacific Design Automation Conference*, pages 109–112, January 1999.

[52] T. Pering, T. Burd, and R. Brodersen. The simulation and evaluation of dynamic voltage scaling algorithms. In *Proc. International Symposium on Low Power Electronics and Design*, pages 76–81, August 1998.

[53] Q. Qiu, Q. Wu, and M. Pedram. Dynamic power management of complex systems using generalized stochastic petri nets. In *Proc. Design Automation Conference*, pages 352–356, 2000.

[54] G. Quan and X. S. Hu. Energy efficient fixed-priority scheduling for real-time systems on variable voltage processors. In *Proc. Design Automation Conference*, pages 828–833, 2001.

[55] A. Rjoub, S. Nikolaidis, O. Koufopavlou, and T. Stouraitis. An efficient low-power bus architecture. In *Proc. of IEEE Int. Symposium on Circuits and Systems*, pages 1864–1867, June 1997.

[56] F. Sanchez and J. Cortadella. Time-constrained loop pipelining. In *Proc. International Conference on Computer-Aided Design*, pages 592–596, November 1995.

[57] M. Sgroi, M. Sheets, A. Mihal, K. Keutzer, S. Malik, J. Rabaey, and A. Sangiovanni-Vincentelli. Addressing the system-on-a-chip interconnect woes through communication-based design. In *Proc. Design Automation Conference*, pages 667–672, June 2001.

[58] D. Shin, J. Kim, and S. Lee. Low-energy intra-task voltage scheduling using static timing analysis. In *Proc. Design Automation Conference*, pages 438–443, June 2001.

[59] Y. Shin and K. Choi. Power conscious fixed priority scheduling for hard real-time systems. In *Proc. Design Automation Conference*, pages 134–139, June 1999.

[60] Y. Shin, K. Choi, and T. Sakurai. Power optimization of real-time embedded systems on variable speed processors. In *Proc. International Conference on Computer-Aided Design*, pages 365–368, November 2000.

[61] R. Sims. Signal to clutter measurement and atr performance. In *Proc. of the SPIE - The International Society for Optical Engineering*, volume 3371, pages 13–17, April 1998.

[62] T. Simunic, L. Benini, and G. De Micheli. Event-driven power management of portable systems. In *Proc. International Symposium on System Synthesis*, pages 18–23, 1999.

[63] A. Sinha and A. Chandrakasan. Operating system and algorithmic techniques for energy scalable wireless sensor networks. In *Proceedings of the 2nd International Conference on Mobile Data Management*, January 2001.

[64] M. Srivastava, A. Chandrakasan, and R. Brodersen. Predictive system shutdown and other architectural techniques for energy efficient programmable computation. *IEEE Transactions on VLSI Systems*, 4(1):42–55, March 1996.

[65] M. Stan and W. Burleson. Bus-invert coding for low-power I/O. *IEEE Trans. on VLSI Systems*, 3(1):49–58, March 1995.

[66] H. Stone. Mars pathfinder microrover: A low-cost, low-power spacecraft. In *Proc. the 1996 AIAA Forum on Advanced Developments in Space Robotics*, August 1996.

[67] Texas Instruments. IEEE 1394 products: Integrated devices, link layer controllers and physical layer controllers. In *http://www.ti.com/sc/1394*, 2002.

[68] A. Wang and A. Chandrakasan. Energy efficient system partitioning for distributed wireless sensor networks. In *Proc. IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 905–908, May 2001.

[69] E. F. Weglarz, K. K. Saluja, and M. H. Lipasti. Minimizing energy consumption for high-performance processing. In *Proc. Asian and South Pacific Design Automation Conference*, pages 199–204, 2002.

[70] G.-Y. Wei, J. Kim, D. Liu, S. Sidiropoulos, and M. Horowitz. A variable-frequency parallel I/O interface with adaptive power-supply regulation. *IEEE Journal of Solid-State Circuits*, 35(11):1600–1610, November 2000.

[71] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced CPU energy. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 13–23, 1994.

[72] W. Wolf. An architectural co-synthesis algorithm for distributed embedded computing systems. *IEEE Transactions on VLSI Systems*, pages 218–229, June 1997.

[73] Q. Wu, Q. Qiu, and M. Pedram. An interleaved dual-battery power supply for battery-operated electronics. In *Proc. Asia and South Pacific Design Automation Conference*, pages 387–390, January 2000.

[74] F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced CPU energy. *IEEE Annual Foundation of Computer Science*, pages 374–382, 1995.

[75] T. Z. Yu, F. Chen, and E. H.-M. Sha. Loop scheduling algorithms for power reduction. In *Proc. IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 3073–6, May 1998.

[76] Y. Zhang, W. Ye, and M. Irwin. An alternative architecture for on-chip global interconnect: segemented bus power modeling. In *Conf. Record (Signals, Systems & Computers) of 32nd. Asilomar Conf.*, pages 1062–1065, 1998.

Appendix A

Tool

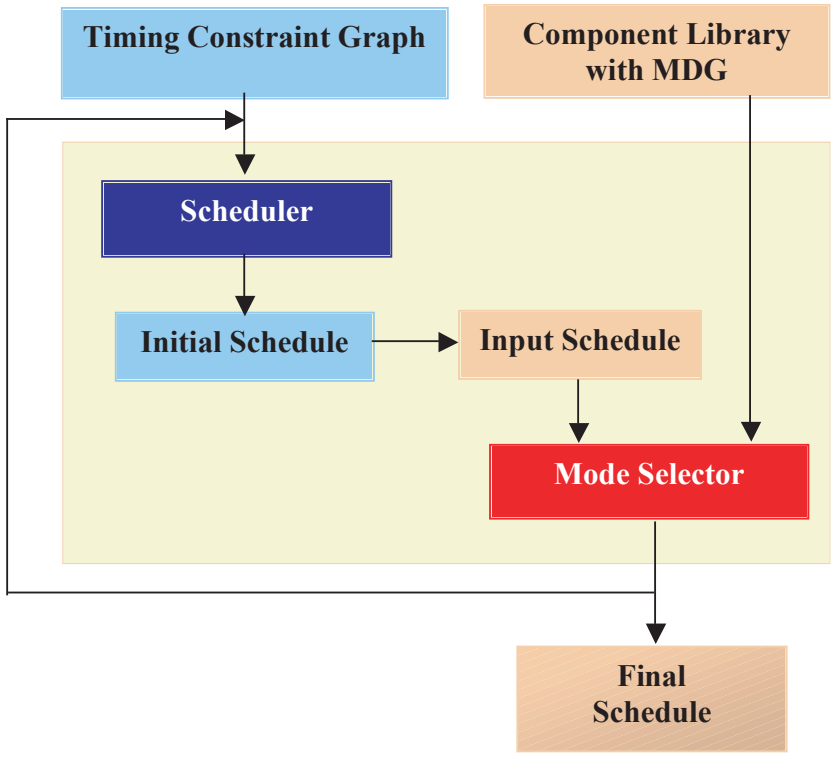
A.1 Introduction

A.1.1 An Overview of IMPACCT tool

IMPACCT tool consists of *Scheduler* and *Mode Selector*. It performs power-aware scheduling and mode selection in order to ensure that all timing/power constraints of the system are satisfied and that all overheads are taken into account. The tool combines the state-of-the-art techniques at the system level, saving designers from many possible pitfalls of system-level power management. The goal of the tool is to expand the range of power/performance trade-offs, so that the system can most effectively adapt to the wide range of power availability in different operating scenarios.

The integrated view of the tool is shown in Figure1. The **inputs** are the *Timing Constraint Graph* and Component Library with Mode Dependency graph, both of which should be manually extracted from the system specification. The scheduler and mode selector work together with a feed back loop, converging the initial raw schedule to the final optimal schedule. The output is the final schedule that considers all the power and timing constraints, and includes feasible mode selection that saves the total system

energy consumption.



*MDG = Mode Dependency Graph

A.1.2 Features of IMPACCT tool version 1.0

- The tool is cross-platform, though examples in this appendix were run on MAC OS X.
- The current version of the tool is not fully integrated yet. In other words, the Scheduler and the Mode Selector are two separate tools. Therefore, the output of the Scheduler has to be manually converted into the input format of the Mode Selector. This procedure will be automated in the future version.
- The Scheduler tool performs a battery simulation in addition to scheduling, which enables to see the battery behavior corresponding to a specific schedule.

This appendix will explain Scheduler and Mode Selector as two independent tools.

A.2 Scheduler

A.2.1 Software Installation

Python

You need Python 2.0.1 or higher to run the tool. Python can be obtained for free from http://www.python.org. Find an appropriate package for your platform (Mac, Windows, Unix, or Linux), follow their instructions to unpack and install on your machine.

Jython

You need Jython 2.0 or higher to run the Scheduler. Jython could be downloaded for free from http://www.jython.org/download.html. Find an appropriate package for your platform (Mac, Windows, Unix, or Linux), follow their instructions to unpack and install on your machine.

Scheduler

1) Download the Scheduler from

   http://embedded.ece.uci.edu/cgi-bin/cvsweb.cgi/tool/scheduler/scheduler.tar.gz

2) Unpack by typing (in the system shell)

   ```
   %tar zxvf scheduler.tar.gz
   ```

   For the older versions of tar, you should type,

   ```
   %gzcat scheduler.tar.gz | tar xvf scheduler.tar
   ```

3) Type the following to compile the battery simulator, which is written in fortran.

   ```
   %cd batsim
   %./compile.bat
   ```

A.2.2 Getting Started

There are several ways to start the scheduler. The choice depends on the usage of *daemon* for running the scheduler and the battery simulator. The scheduler can either run on a built-in Jython, or on a daemon. However, the battery simulation MUST run on a daemon. The location of the daemons could be local or remote. The following are three typical ways, though other combinations are also possible.

1) Running the scheduler as a built-in method, and running the battery simulation on a localdaemon. (default)

2) Running both on local daemon

3) Running both on remote daemon

The following is the explanation of each method. Choose one of them, and follow the steps.

1) This is the simplest way. Open the shell, go to the directory where you unpacked the scheduler, and type

```
%python run.py [-x] sd bd gui
```

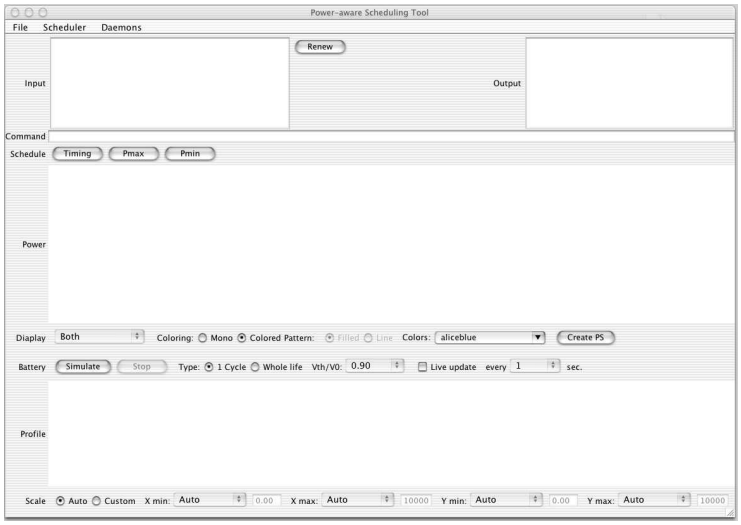
`sd` : Scheduler daemon

`bd` : Battery simulation daemon

`gui` : GUI

`-x` : If current terminal window supports xterm, the specified daemon(s)/GUI will be started in new xterms. If not, it is preferred to start each daemon/GUI in a separate terminal window without -x switch.

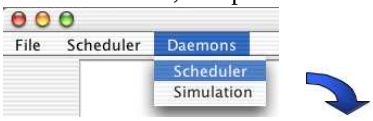
The Scheduler GUI will pop up.

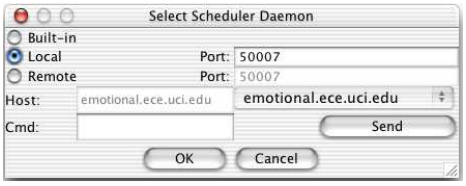


2) Open the shell, go to the directory where you unpacked the scheduler, and type

```
%python run.py [-x] sd bd gui
```

When the Scheduler pops up, open Daemon→Scheduler. Choose Local, specify the Port number, and press OK.



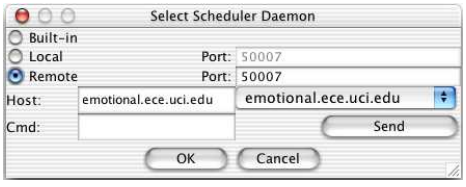


Running the programs on local daemon could be faster than running on built-in Jython.

3) Open the shell, go to the directory where you unpacked the scheduler, and type

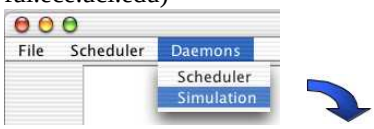
```
%python run.py [-x] sd bd gui
```

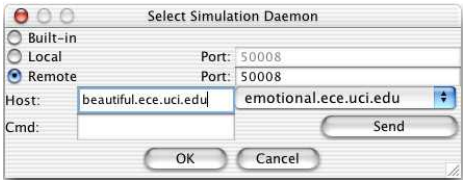
When the Scheduler pops up, open Daemon→Scheduler. Choose Remote, specify the Port, type in the Host or select from the list, and press OK. If you type in the host, then the selected list is disabled.



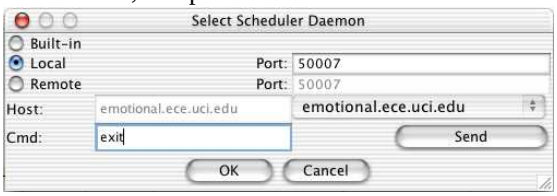
Then, open Daemon→Simulator. Choose Remote, specify the Port, type in

the Host or select from the list, and press OK. If you type in the host, then the selected list is disabled. (The example below specifies the host as beautiful.ece.uci.edu)





Note: When you are running on Daemon, you must remember to close it when you finish running the tool, before closing the Scheduler GUI. Otherwise, the Daemon will run forever. In order to close a daemon, type exit on the command box(Cmd), press Send button, and press OK.



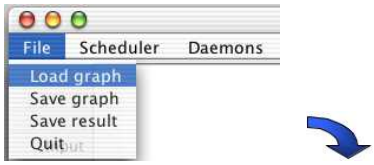
A.2.3 Running the tool

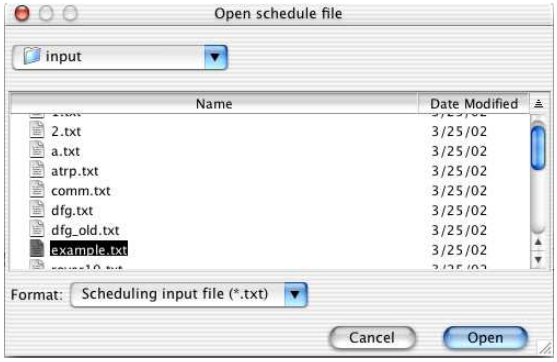
Loading the Input

There are two possible ways to load the input. You can (1) load it from file or (2) directly type in the input.

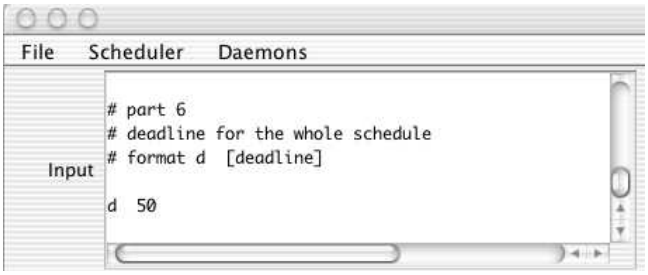
(1) If you want to load the input from file, you must first create an input text file that that contains the representation of timing constraint graph. See Section A.4 to get the reference of input format. Save your input file in the directory named input, which you will find under the directory where you unpacked the tool.

Go back to the Scheduler GUI and open File→Load graph. Choose your input file and click on Open.





You will see your input file loaded in the input text box on Scheduler GUI.



(2) You can also directly type the input into the input text box on Scheduler GUI.

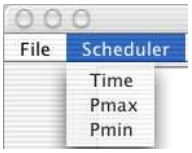
Execution

(1) Scheduler

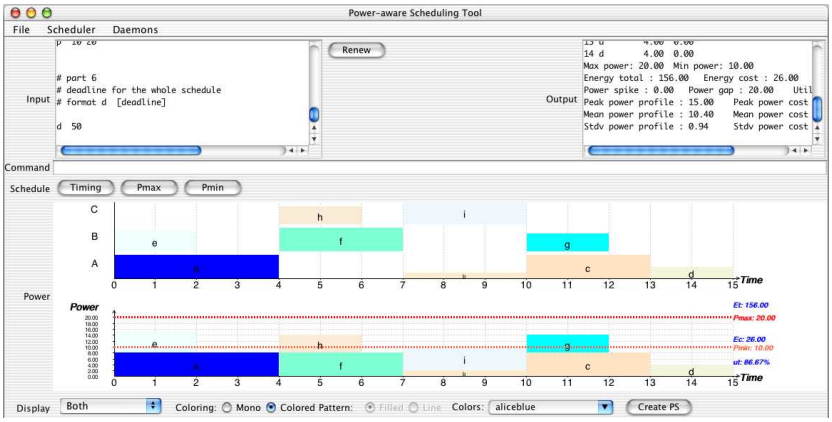
When the input is loaded onto the Input box, you can run three programs that output schedules meeting: i) Timing constraint, ii) Max power constraint, iii)

Min power constraint, respectively. In order to execute, either click on each corresponding button,

or select from the menu.



After running three programs sequentially, you will get the output schedule. The graphical result will be shown in the middle box labeled Power, and the output text will be shown on the Output text box on the top right side.



- E_t: Total energy
- E_c: Energy cost (Area of boxes above the minimum power constraint)
- P_{max}: Maximum power constraint
- P_{min}: Minimum power constraint
- U_t: Resource Utilization

(2) Battery simulator

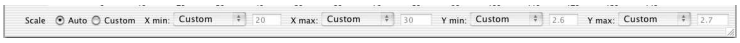
After running the scheduler, you can run the Battery simulator.

i) Specify the feature of simulation on GUI.

- 1 Cycle simulates only one period time of the schedule, whereas Whole Life simulates the whole battery life.
- Vth/V0 is the ratio of the threshold voltage to the initial output voltage of the battery. This ratio determines the length of the profile, since the simulation continues until the Voltage reaches the threshold voltage.
- If you check Live update and specify the second, you will be able to see the profile at the run-time of simulation. Otherwise, you will see the profile at the end of simulation.

ii) Run the simulator.

Press the Simulate button. You will see the output in the Profile box on the bottom of GUI. On the right end of the graph, the total time of the profile(T) and the threshold voltage(V) will appear. The example below is the profile of the whole battery lifetime, with default Vth/V0 Ratio, 0.9, which was run without live update.



iii) Scale

You can perform custom scale for analysis. Check Custom radio button, and type any minimum and maximum numbers into each text box, and

press Return key on keyboard.

Other features

(1) Display

You can change the display of the graphical output of scheduling.

i) View

- Time - This option shows the component level graph of the schedule and power consumption.
- Power - This option shows the system level graph of the schedule and power consumption.
- Both - This option shows both graphs.
- T & P-curve - This option shows a graph that combines both graphs. On the Time graph, the Power graph is overlapped as an outline curve.

ii) Color scheme

You can change the color scheme of the Power graph. Mono enables other options of color scheme.

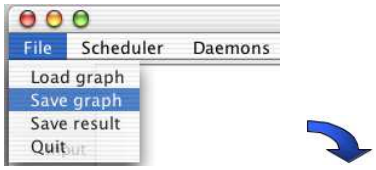
- You can specify a color from the color list.
- Choosing Filled fills the whole block with the specified color, which makes it easier to see the total energy consumption.
- Choosing Line shows only a silhouette of the power consumption.

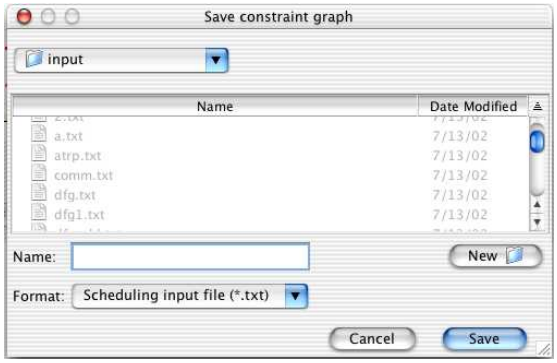
(2) Reload

You can load a new input and re-execute. Load the input with the same method as before, and press the Reload button, which is next to the Input box on GUI. Then, repeat the steps explained in the previous section.

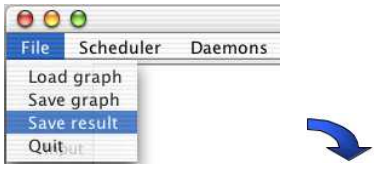
Saving the result

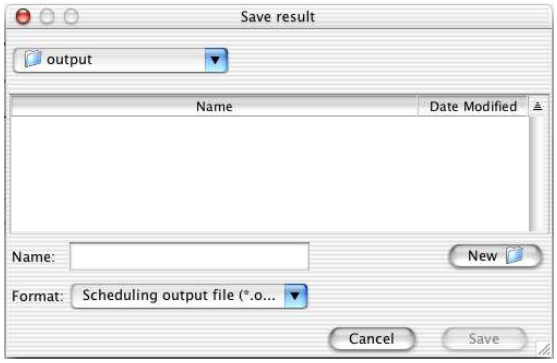
To save the current timing constraint graph(the input), open File→Save graph, and save the file.





To save the output as a text file, open File→Save result, and save the file.





To save the output graphs as PostScript, press the Create PS button on the GUI. This will automatically create two files under output directory.

- [file]_sch.ps, this is the .ps for the scheduling result. If the input file is ”example.txt”, this file will be ”example_sch.ps”.
- [file]_sim.ps, this is the .ps for the simulation result. If the input file is ”example.txt”, this file will be ”example_sim.ps”.

Note: Do not forget to close the daemon, before closing the GUI.

A.3 Mode Selector

A.3.1 Software Installation

Python

You need Python 2.0.1 or higher.

Tkinter

To display the graphical interface, the Mode Selector uses the Tkinter module in Python.

To test whether the Tkinter module is properly installed, open the shell, and type

```
%python
%import Tkinter
```

As an alternative way, you can run either python.exe or idle.pyw by double clicking the icon, and type

```
>>>import Tkinter
```

If no error message comes up, then the Tkinter is properly installed.

Mode Selector

1) Download the Mode Selector from

   http://embedded.ece.uci.edu/cgi-bin/cvsweb.cgi/tool/modesel/ms071302.tar.gz

2) Unpack by typing (in the system shell)

   ```
   %tar zxvf ms071302.tar.gz
   ```

   For the older verson of tar, type

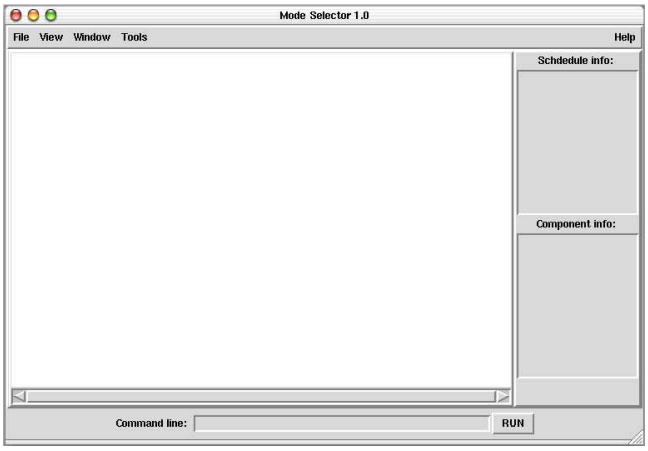
   ```
   %gzcat ms071302.tar.gz | tar xvf ms071302.tar
   ```

A.3.2 Getting Started

To start the Mode Selector, open the shell, go to the directory where you unpacked the Mode Selector, and type

```
%python ms_gui.py
```

The Mode Selector window will pop up.



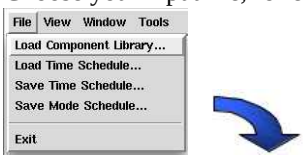
A.3.3 Running the tool

Loading the input

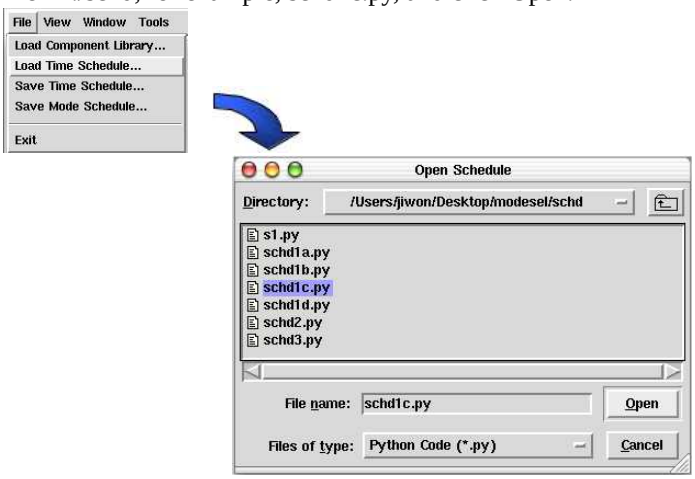
Before running the tool, two input files must be saved in the same directory the mode selector is installed. These are the Component library with MDG, and the input schedule. The input schedule is the output file of the Scheduler with converted file format(.txt→.py). The input formats could be found in [2] in Section A.4.

When the input is ready, go back to the Mode Selector GUI.

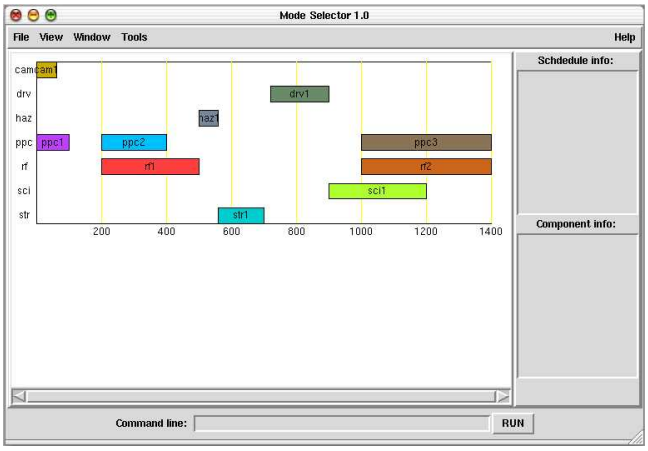
To load the Component Library with MDG, open File→Load Component Library. Choose your input file, for example, lib1.py, and click Open.



To load the initial schedule, open File→Load Time Schedule. Choose your input file in ./schd, for example, schd1c.py, and click Open.

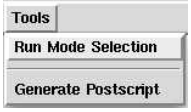


You will see the input graph with random color scheme, loaded on your GUI.

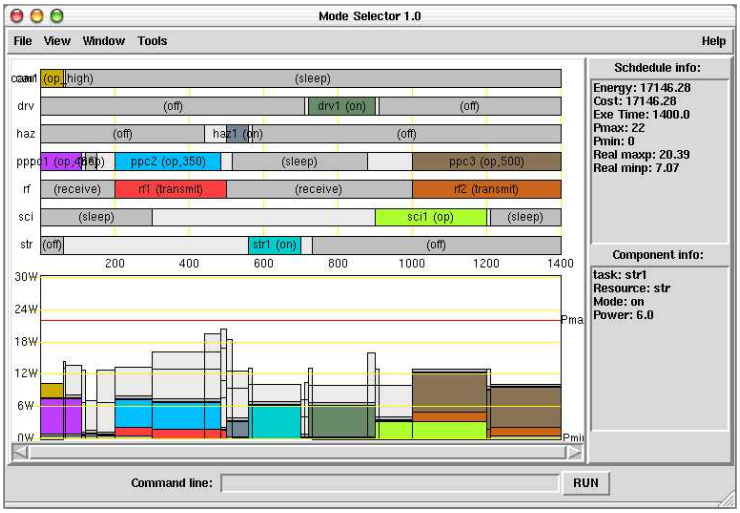


Execution

To run the Mode Selector, select Tools→Run Mode Selection.



You will see the result on your GUI. It contains (1) the mode schedule graphs on the left, (2) schedule information on the top right, and (3) component information on the bottom right.



(1) The Mode Schedule graphs

The upper graph shows the time view of mode schedules of each component. The boxes with chromatic colors are the tasks, labeled with task name and mode name. The gray boxes are idle intervals, labeled by the mode name. The light gray boxes are the mode change intervals, which have no label, and the mode is non-determined.

The lower graph shows the power profile, which is the time view of total power consumption. The red lines show the maximum and minimum power constraints. (In the picture above, the minimum power constraint is zero, so the line is on the

bottom)

(2) Schedule info

It shows the energy consumption(Joule), energy cost(Joule), total execution time(sec), maximum power constraint(Watt), minimum power constraint(Watt), real maximum power(Watt), and real minimum power(Watt). The energy cost is the energy consumption above the minimum power constraint. Real maximum and minimum powers are the peak powers that the mode schedule reaches.

(3) Component info

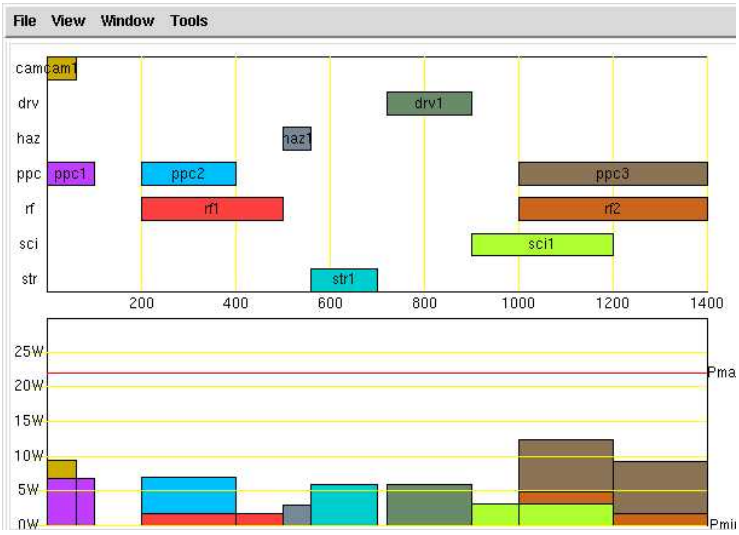
The information of the tasks appears on this box dynamically, as you move around the mouse on Mode Selector GUI, and point to any component. It shows the task name, allocated resource, mode, and the power consumption(Watt).

Other features

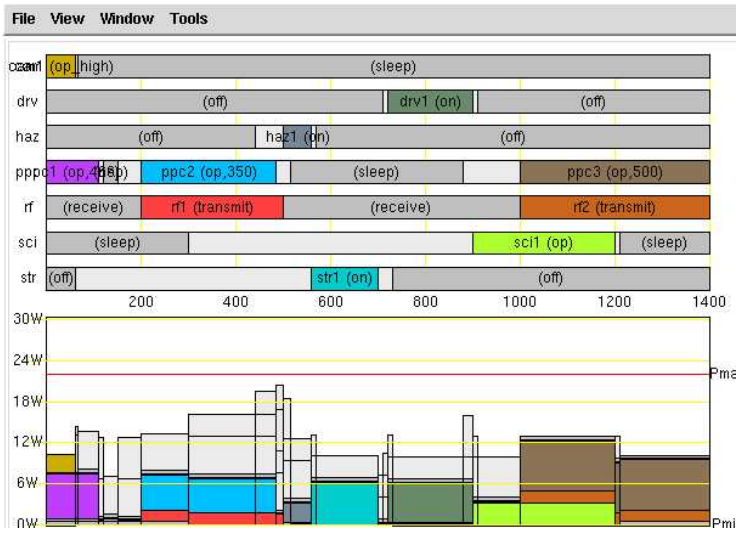
(1) You can change the view of the result with three options: Task Only, Show All, and Power profile.

(Note: Change color scheme and Refresh are not implemented yet.)

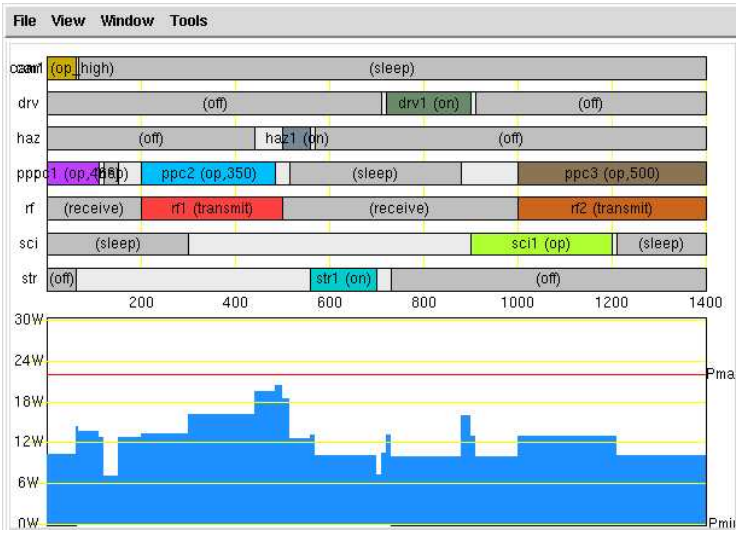
– Task Only shows the result excluding the idle and mode change intervals.



– Show All gives the original output graph, including all intervals.



– Power profile gives the monochromatic view of power, which makes it easier to see the energy consumption and the fluctuation of power.



(2) You can also disable and enable the view of the graphs by selecting among window options.

- Time window shows the upper graph
- Power window shows the lower graph
- Both shows both(default).

Saving the result

You can save the output graphs in PostScript format. Select Tools→Generate Postscript, choose the desired directory (default is schd), type the file name and press Save.

Note: Saving the input and output as text files will be implemented in the next version.

A.4 Input formats

The Timing Constraint Graph is represented in a text file. You can find example.txt in the package as an example. Component library with MDG and the Input schedule are represented as a dictionary data structure in Python. lib1.py and schd1c.py are the examples, respectively. More information about the inputs Timing Constraint Graph and Mode Dependency Graph could be found in the reference papers.

The following are some partial examples for the demonstration of input formats. For complete examples, please look into the example files.

Input for the Scheduler

```
# This file is the timing constraint description
# of a scheduling problem.
# It is used as the input file to the IMPACCT scheduler.

# part 1
# the header of the constraint graph description
# format:   graph [name]

graph  test



# part 2
# resources
# format:   r    [resource ID] [1 (currently not used)]

r   A   1
r   B   1
r   C   1
```

```
# part 3
# tasks
# format:   t   [task ID] [delay] [power] [resource ID]

t   a   4   8   A
t   b   3   2   A
t   c   3   8   A
t   d   2   4   A
t   e   2   7   B
t   f   3   8   B
t   g   2   6   B
t   h   2   6   C
t   i   3   8   C


# part 4
# timing constraint
# format:   c   [event1 ID] [event2 ID] [constraint]
# event can be start and end of a task,
# represented by task.s and task.e

c   a.s b.s 7
c   a.s c.s 8
c   b.s g.s 3
c   b.s i.s 3
```

```
c   c.s d.s 2
c   d.s g.s -6
c   e.s b.s 3
c   f.s d.s 4
c   g.s d.s 2
c   h.s c.s 5
c   h.s i.s 2


# part 5
# system-level power constraint
# format:   p   [min power] [max power]

p   10  20


# part 6
# deadline for the whole schedule
# format:   d   [deadline]

d   50
```

Input for the Mode Selector: Component library

```
# ’c_componentname’ is a dictionary for a component
# imode stands for idle modes
# wmode stands for working modes
# however, all working modes apply for the idle time also.
#
```

```
# power is the power consumption of the mode, unit is Watt.
# speed is the performance of the component in Spec95int number
# cost is the mode change overhead,
# both time and power overhead are included


c_mc = {
    'imode': ['sleep', 'idle'],
    'wmode': ['op,30', 'op,50', 'op,75', 'op,100'],
    'power': {
        'sleep':    1.6e-4,
        'idle':      0.05,
        'op,30':    0.1,
        'op,50':    0.2,
        'op,75':    0.3,
        'op,100':   0.4
        },
    'speed': {
        '30':   4.1,
        '50':   6.0,
        '75':   9.7,
        '100':  14.2
        },
    'cost' : {
        ('sleep', 'op'):    (10, 0.4),
        ('sleep', 'idle'):  (5, 0.05),
        ('idle', 'op'):     (3, 0.4),
```

```
        ('op', 'idle'):     (1, 0.4),
        ('op', 'sleep'):    (1, 0.4)
        }
    }


# time-variant resource/mode functions
# T is the temperature

functions = {
    ('drv', 'on'):
        '-0.1225 * T + 1.0',
    ('drv', ('off','on'), 'time'):
        '(-1.875 * T +10)*( T < 0) + 10*( T >=0)',
    ('str', 'on'):
        '-0.09 * T + 2.4'
    }


# time to temperature mapping

# use interpolation to obtain the intermediate values

temperatureProfile = {
    0:  0.0,
    300:    -20.0,
    500:    -40.0,
```

```
    800:    -40.0,
    1000:   -60.0,
    1400:   -80.0
    }


# component instance name: component type name

components = {
    'cam':  c_cam,
    'drv':  c_drv,
    'haz':  c_haz,
    }

colorlib = [
    'AQUAMARINE3',
    'AQUAMARINE4',
    'BISQUE4',
    'BLUE',
    ]
```

Input for the Mode Selector: Input schedule

```
schedule = {
    'haz1.s':   500,
    'haz1.e':   560,
    'str1.s':   560,
    'str1.e':   700,
```

```
    'drv1.s':   720,
    'drv1.e':   900,
    'cam1.s':   0,
    'cam1.e':   60,
    'ppc1.s':   0,
    'ppc1.e':   100,
    'ppc2.s':   200,
    'ppc2.e':   400,
    'ppc3.s':   1000,
    'ppc3.e':   1400,
    'rf1.s':    200,
    'rf1.e':    500,
    'rf2.s':    1000,
    'rf2.e':    1400,
    'sci1.s':   900,
    'sci1.e':   1200
    }

# allocation is a mapping from task to resource
allocation = {
    'ppc1': 'ppc',
    'ppc2': 'ppc',
    'ppc3': 'ppc',
    'rf1':  'rf',
    'haz1': 'haz',
    'cam1': 'cam',
    'drv1': 'drv',
```

```
    'str1': 'str',
    'rf2':  'rf',
    'sci1': 'sci'
    }

# functional modes setting
functional_mode_settings = [('cam1', 'op_high')]

# functional modes constraints
functional_mode_constraints = {
    ('cam1', 'op_high'): [
        ('ppc1', 'op,500'),
        ('ppc1', 'op,466'),
        ('ppc1', 'op,450'),
        ('ppc1', 'op,433'),
        ('ppc1', 'op,400'),
        ('ppc1', 'op,375'),
        ('ppc1', 'op,366'),
        ('ppc1', 'op,350')
        ],
    ('rf1', 'transmit'): [
        ('ppc1', 'op,500'),
        ('ppc1', 'op,466')
        ]
    }

# timing constraints
```

```
timing_constraints = {
    ('cam1.s', 'ppc1.e'):   -200,
    ('ppc2.e', 'sci1.s'):   0,
    ('ppc2.e','rf1.e'): 0
    }

deadline = 1400
maxPowerConstraint = 22
minPowerConstraint = 0
```